Back to BASIC

John G. Kemeny
Thomas E. Kurtz

Creators of BASIC

The History, Corruption, and Future of the Language

Back to BASIC

John G. Kemeny
Thomas E. Kurtz

Creators of BASIC

The History, Corruption, and Future of the Language

Twenty years ago, John Kemeny and Thomas Kurtz created BASIC, now the most popular programming language in the world. Today, they would just as soon disown the result of their work.

> "We are greatly concerned that a generation of students is growing up learning Street BASIC, an illiterate dialect of a lovely language. . . . There have been devastating criticisms of BASIC in the literature. Unfortunately as it applies to Street BASIC, we agree with them."

Discussing their latest version of the language, True BASIC,™ Kemeny and Kurtz make a bold case that reestablishes BASIC's value, challenges the assumptions of many in the field, and shows why True BASIC will be the universal choice among the world's next generation of microcomputer users.

BACK TO BASIC begins with the inception of the Beginner's All-purpose Symbolic Instruction Code and its philosophical and educational foundations. It traces the history of the language, its corruption, and introduces structured programming in True BASIC. Kemeny and Kurtz show that BASIC is as "serious" as any computer language available today, and they provide examples of applications that are both sophisticated and easy.

Professors **John G. Kemeny** and **Thomas E. Kurtz** together invented the original BASIC language and, with several partners, the new True BASIC language. Professor Kemeny, chairman of the Mathematics Department at Dartmouth College, was president of Dartmouth for eleven years, a research assistant to Albert Einstein, and chairman of President Carter's commission on Three Mile Island. Professor Thomas E. Kurtz heads the American National Standards Institute committee on standards for the BASIC language.

Cover design by Marshall Henrichs

ADDISON-WESLEY PUBLISHING COMPANY, INC.

ISBN 0-201-1

BACK TO BASIC

The History, Corruption, and Future of the Language

John G. Kemeny
and
Thomas E. Kurtz

Addison-Wesley Publishing Company, Inc.
Reading, Massachusetts · Menlo Park, California
Don Mills, Ontario · Wokingham, England
Amsterdam · Sydney · Singapore · Tokyo · Mexico City
Bogotá · Santiago · San Juan

True BASIC is a trademark of True BASIC, Inc., Hanover, New Hampshire.

Library of Congress Cataloging in Publication Data

Kemeny, John G.
Back to BASIC

Includes index.
1. Basic (Computer program language) I. Kurtz, Thomas E. II. Title. III. Title: Back to B.A.S.I.C.
QA76.73.B3K44 1985 001.64'24 84-24546
ISBN 0-201-13433-0

Copyright © 1985 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the Publisher. Printed in the United States of America. Published simultaneously in Canada.

Set in 10 pt Century Schoolbook by Ampersand Publisher Services, Inc., Rutland, VT

First printing April 1985
ABCDEFGHIJ-HA-898765

BACK TO BASIC

The History, Corruption, and Future of the Language

INTRODUCTION

In 1963 Dartmouth College made a decision to enable all students to become computer-literate. To implement that decision required sweeping new ways to bring computing and students together. Two of those ways stand out. The first was that we needed a totally different way to distribute the computer resource; batch processing simply would not work. The second was that we needed to develop a new language, one that was far easier to learn and use for large numbers of students; none of the existing languages was suitable.

To meet these needs, the two of us, with the help of a group of exceptionally able undergraduates, developed the first fully functional general-purpose time-sharing system. We also developed a new language, one that was ideal for introducing beginners to programming and yet could serve as a language for all applications, even for large and complex software systems.

The new language was BASIC. Its wide acceptance went far beyond anything we had hoped for. That was the good news. The bad news was that its very success led to problems. There developed a proliferation of BASICs, ranging from the excellent to the terrible. If we are the parents of BASIC, there are some descendants we would frankly like to disown!

At Dartmouth we had a clear philosophy about the language. As it grew from modest beginnings to the "all-purpose" language we intended, we did not lose sight of the original principles. The name BASIC stood for Beginner's All-purpose Symbolic Instruction Code. The first word of that

acronym is as relevant today as it was then. No matter how powerful the language became, we never forgot the needs of beginners.

We worked hard to make it truly general-purpose, while keeping the original clean and simple design. When computer science made the case for fully structured languages, we and our colleagues redesigned and expanded the structures in BASIC. When graphics terminals became available, we added a powerful and easy-to-use graphics package. As a result, over 90 percent of the Dartmouth students became computer-literate (that is, able to write simple programs), the computer became essential to hundreds of courses, and BASIC became the language of choice in a highly diverse academic community.

The fate of BASIC in the outside world was not so fortunate. The great improvements that were taken for granted here, and at some other institutions, did not make their way into most commercially available versions of the language. And when microcomputers appeared, first with very limited capabilities, software houses made severe compromises in order to fit BASIC into very small memories. Microcomputers are now much larger and faster even than the computer on which BASIC was born back in 1964, but the compromises have become "features," and they were kept—no matter how ugly.

Worst of all, the original philosophy that led to BASIC was lost sight of. For example, we have always believed that users should be protected from the peculiarities of the hardware, so that BASIC programs could be run on all kinds of machines. Instead, there are now several versions of BASIC on each microcomputer. It is even true that versions written by the same firm, for different computers, are grossly incompatible. And even the best of these tend to be roughly vintage 1966 BASIC.

Today the world doesn't have to be sold on the importance of computing and computer literacy. But the world has not yet fully learned what we were able to demonstrate two decades ago—that a simple, well-designed language and a simple computer interface can allow the many to know what only a few could know before.

Delivering computer power is no longer a problem—personal computers do it so much better than the crude time-sharing system we developed years ago. But computer language still remains a problem. We decided that once more we had to step in and take charge. We believe that we can construct a powerful, modern, easy-to-use BASIC for microcomputers, and still adhere to the tried and true principles we set down years ago. This new BASIC is called True BASIC. Written with the aid of three systems programmers who have a long association with Dartmouth, it will be available on most of the newer microcomputers.

Once again it will be possible to have the same program run on a variety of machines.

Over the past decades we have often been urged to recount the history of the language, why we created it, and what principles distinguish it from earlier computer languages. Although one of us (Kurtz) did write an account for experts,[1] we have never told the story to a general audience. During the intervening years other commitments made such a project impossible. Tom Kurtz served as the first director of the Kiewit Computing Center at Dartmouth and was heavily involved in regional and national computing projects. John Kemeny served as president of Dartmouth College for eleven and a half years. With the appearance of True BASIC, now seems like a good time to tell the full story.

This book is the story of BASIC, from its elementary version in 1964 to True BASIC in 1984.

The first three chapters of the book are a history of BASIC, from birth, to its growth into a powerful language, to the development of a fully structured modern language. It is a personal history, and we recount many details not previously published. We have tried to recall why we did what we did, what decisions turned out to be fortunate, and where we erred.

Chapter 4 discusses what went wrong. Unfortunately, a great deal did. If the reader cannot resist a quick peek at that chapter, we promise not tell anyone!

Chapter 5 is aimed at those millions who have programmed in some version of BASIC but have never seen the kind of BASIC we have used at Dartmouth for nearly a decade. It introduces the reader to structured programming through carefully chosen examples. We make the case that the transition from a sloppy version to the fully structured one is not difficult. And once one makes the transition, one will never revert to old habits. We know, because we had to go through this transition ourselves. The result is that long programs are much easier to write, to debug, and to read.

In chapter 6 we attempt a factual comparison with some of the more popular computer languages. Our aim is to show that structured BASIC deserves to be ranked as an equal among other "serious" languages. It is still one of the easiest languages to learn, and yet it has all the features one would want for introductory courses in programming or computer science, or for large application packages.

Chapter 7 is there purely for the fun of it. We hope by then to have whetted the reader's appetite for structured BASIC, and want to share a variety of examples. The examples are, we trust, interesting in them-

selves. And they illustrate the ease with which a wide variety of applications can be programmed.

JOHN G. KEMENY

THOMAS E. KURTZ

September 10, 1984

BACK TO BASIC

The History, Corruption, and Future of the Language

CONTENTS

Introduction **vii**

1: The Birth of BASIC **1**

2: BASIC Grows Up 1964–1971 **19**

3: BASIC Matures 1971–Present **39**

4: What Went Wrong? **55**

5: Structured Programming **67**

6: True BASIC and Other Languages **89**

7: Further Examples **107**

Notes **135**

Index **137**

1
THE BIRTH OF BASIC

Our memories of the birth of BASIC are still vivid after more than two decades. On most points our recollections agree, and we will share these with you. In a few cases our memories do not agree or the recollections are very personal. In these cases each will speak for himself.

THE BAD OLD DAYS

Let us recount what the normal use of computers was in 1963. Computers in the modern sense had existed since the early fifties, but they were large beasts and very expensive. Computer center directors considered it their major task to protect machines from human inefficiency. Since computers were so fast and so expensive, while human beings were slow (and cheap?), schedules were designed to maximize the amount of computing carried out by a machine.

The human user punched a program on IBM cards and submitted it to an operator. The operator collected hundreds of such jobs and fed them to the computer in a "batch." The machine worked on one task at a time, until it was completed, and then printed results on a not-very-fast printer. When all the jobs in the batch were completed, it was time for the next feeding of the computer.

The user returned the next day to pick up the results. Since the

computer could carry out in seconds what would take a human being weeks or months, it seemed reasonable to wait twenty-four hours for the results. And so "batch processing" was universally accepted.

It might indeed have been a good system if the user received a solution the next day. But what he or she found on the printout was something like "Illegal instruction on card 27." And then the long process of finding errors, or "debugging," began. The first few days were needed just to correct keypunching errors.

KEMENY: One of the happiest days in my life was when I realized that I would never have to punch another card! The keypunch was an awkward device, and any punching error required tearing up the card, and starting over, since there was no reasonable way of plugging up the little holes.

Then one day there was not even an error message, just a blank piece of paper. Perhaps the computer had calculated the answer to the problem, but the user forgot to specify that the answer should be printed, so it wasn't! An additional card was hastily inserted—too hastily—and the result was twenty printed pages jammed full of numbers, when you expected a one-line answer. And so it continued until, about two or three weeks later, you actually obtained a solution to your problem.

KEMENY: I first became aware of the shortcomings of batch processing in the summer of 1956, as a consultant to the RAND Corporation (Santa Monica, California). I saw highly paid experts stand in line for hours to get a five-second debugging shot. I left behind a recommendation that a system be devised to allow brief debugging interruptions to the batch system. But I had not yet faced the need to change the system of computer use completely.

KURTZ: I remember one of my early experiences at Massachusetts Institute of Technology, in 1956 or '57, with batch processing. I was then a poorly paid instructor and research associate, so it was of no consequence that it took several months to solve my problem (I visited MIT once every two weeks, by train). But it was a shock to realize that I had used up more than an hour's worth of very valuable computer time on the IBM 704! It must

have been all those "memory dumps" I took to study between visits to MIT.

As we began to plan a computer system for Dartmouth, it became clear that these old ways would not work. Scientists and engineers might be willing to put up with such service, but we were convinced that Dartmouth students would rebel.

A BETTER IDEA

There were some experiments going on (notably at MIT and Bell Labs) that would allow a number of people to use the same computer at the same time. The great speed of the computer would allow all of them to receive fast service. And although people would make just as many mistakes, error messages would appear within seconds, and one could get a program debugged in fifteen minutes to half an hour. It was expected that the system would be less efficient but much kinder to human beings. The machine waited on human convenience, not the reverse! And the ultimate irony would be that these user-friendly systems would prove *more* efficient than batch processing. As Kurtz recalled, with batch processing users tended to ask for enormous and costly "memory dumps" that were totally unnecessary if one received fast response and could try a correction immediately.

KURTZ: We started thinking about time-sharing in the early sixties. While I was visiting John McCarthy at MIT, he told me, "You guys ought to do time-sharing." I went back to Dartmouth, said the same thing to John Kemeny, and he instantly agreed. There simply was no question in either of our minds.

KEMENY: It was in 1962, while I was chairman of the Dartmouth Mathematics Department, that my colleague Tom Kurtz came to me with an outrageous suggestion. He started by asking, "Don't you think the time is approaching when every student should learn how to use a computer?" And I said, "Sure, Tom, but it isn't physically possible to teach so many

students." And then came Tom's radical proposal: "I think we could design a completely different way of using computers that would make it possible to give computer instruction to hundreds of students."

At the time, Dartmouth had only a very small computer. We had already learned that able undergraduates could achieve incredible results with the most primitive computer. But the computer was so small that it could be used by only one student at a time. It was far less powerful and had far less memory than today's personal computers. It therefore had to have quite fancy software if many students were to use it. So we began experimenting.

KURTZ: In one experiment with this computer, which was an LGP-30, we were able to allow about five students to complete their small programs in a total time of fifteen minutes. Each student could have two or three "turnarounds" in that fifteen-minute period. Then another five students would show up. This convinced me that an easy-to-use interactive system could allow hundreds of students to use it and thus learn about computers.

KEMENY: Tom's suggestion—to open computing to all students—was a radical one, way ahead of its time. It would significantly alter the modern history of the college and have a major national impact.

We decided to design and implement such a time-sharing system at Dartmouth. We came up with a totally new computer architecture, in which one computer took care of all communications and scheduling, while the other could concentrate on serving one user at a time. We used two General Electric (GE) computers for tasks their designers never intended, and set Dartmouth on a road that would put it into a pioneering role in computer education.

We were very fortunate to receive full support from President John Sloan Dickey and the Board of Trustees. Dean Leonard M. Rieser, a physicist, was an early and ardent supporter. And the dean of our engineering school was Myron Tribus, one of the most far-sighted people we know. He became a great booster. We believe that they, with the help of a key trustee, were responsible for getting full Dartmouth backing. We

were even allowed to "go GE" when most of the world was pressuring us to "go IBM."

We approached the National Science Foundation (NSF) for the necessary funds (mostly for equipment, and modest for those days). They submitted our application to outside referees. A typical reviewer said that we had no idea how large and difficult a task we were undertaking. We were hopelessly understaffed—two faculty members part-time plus a dozen students. And not even graduate students but undergraduates! To the credit of NSF, they took a chance on us in spite of the negative evaluations. And one year from the start of our project (seven months after the equipment arrived), we were teaching hundreds of students on the Dartmouth Time Sharing System.

The reviewers were right that we didn't know how difficult a job we were undertaking. If we had, we might never have tried. But they were completely wrong about the use of undergraduates. I would later say that we were first to succeed because the others used professionals while we used undergraduates! Undergraduates will work endless hours, are open to new ideas, creative, and willing to take on impossible tasks.

For example, John McGeachie and Mike Busch solved the problem of communication between two completely different GE computers, one of which did not even have a manual since it was never intended to be programmed outside the factory. It was the strangest collaboration we have ever witnessed: two students from very different backgrounds, of different temperament and appearance, each thoroughly identified with "his" computer. They would stand by their machines and yell at the tops of their voices: "Mike, you are not responding!" "Why, you never got through to me!" They may have been the original odd couple, but what they achieved, in the short time they had, was truly astounding.

It was highly appropriate that when the Federation of Information Processing Societies awarded us their first Pioneers' Day award, our students received recognition as full partners.

The coming of time-sharing fundamentally changed the use of computers. Although at the time it would have been prohibitively expensive to give each user his or her own computer, we were able to create the illusion that the large computer was just sitting there waiting to serve. It was the beginning of personalized, interactive computing. The recent arrival of personal computers will carry the availability of individualized computer service to a much larger group of users. But the essential breakthrough occurred in 1964.

KEMENY: In the midst of planning time-sharing, I made a suggestion: "Tom, if we design this great new system, why

don't we also create a really nice language? Surely we can do better than FORTRAN for teaching purposes?" It led to one of the few disagreements I had with Tom. He agreed in principle that we could design a much nicer language, but felt that it might be possible to come up with reasonable subsets of FORTRAN or ALGOL. His worry was that we would teach our students a language that could not be used outside of Dartmouth.

Tom deserves all the credit for proposing the time-sharing project. But I do occasionally remind him of the fears he once had about our new language. The language is BASIC, and it is now the most widely used computer language in the world.

THE ORIGIN OF THE SPECIES

A computer is created speaking one language, called—not surprisingly—"machine language." Its alphabet consists of zeros and ones, corresponding to electronic or magnetic devices being "off" or "on." To make it slightly more palatable, the manufacturer provides an "assembly language," which allows mnemonic codes for frequently used combinations of zeros and ones, and provides certain other shortcuts. Yet assembly language is difficult to learn and use, and most computer scientists agree that one should go to almost any extreme to avoid having to code in assembly language. It is time-consuming, frustrating, hard to read, and very hard to debug.

The appearance of FORTRAN (around 1957) was a major advance in computer programming. It allowed one to write programs in a combination of a few English words and mathematical formulas. The language was designed for numerical applications, particularly for engineers. A much improved version of the language is still widely used today.

But it was intended to be used by experts. It was common to take two months of training in FORTRAN before you tried writing your first program.And the alternatives were no better for our purpose. We wanted a language for a lay audience, one that could be learned easily and whose advanced features could be learned later, when the student was ready for more sophisticated concepts.

The genealogy of BASIC really goes back to 1956. We began

computing on the IBM-704, first at General Electric in Lynn, Massachusetts, and later at MIT. FORTRAN didn't arrive until the following year, so we had to learn SAP (Symbolic Assembly Program), which was assembly language. But would our colleagues be willing to learn SAP if they wanted to do computing? We thought not, and so devised a simplified version of it that we called Darsimco, for Dartmouth Simplified Code.

The idea was to provide templates, or sequences of instructions, for performing most of the simple tasks. For instance, if "A = B + C" was wanted, Darsimco said to use

```
FLD B
FAD C
FST A
```

Persons familiar with assembly language will recognize the sequence as "floating load, floating add, floating store." Our colleagues didn't have to understand what FLD, FAD, or FST meant; all they had to do was to use them as we prescribed.

Darsimco didn't catch on and therefore must be judged a failure. Besides, FORTRAN became available the following year. But the point is that we were both deeply concerned about making computing as simple as possible. Seven years later, in 1963, we finally hit on the way to do it—interactive computing with BASIC.

KURTZ: The experiment I mentioned earlier took place on the LGP-30. Steve Garland and Bob Hargraves, together with classmates Anthony Knapp and Jorge Llacer, had built a compiler for the ALGOL language. Later they constructed a "load and go" version of ALGOL. We called this system Scalp, which stood for Self Contained ALGOL Processor. All the student user had to do was to load his paper tape, and Scalp would either run the program or produce error messages. There was even a crude interactive debugger, so the student didn't have to leave the machine to make trivial corrections.

KEMENY: We experimented both with implementation of standard languages and with the creation of new, easy-to-learn languages. Our students Steve Garland and Bob Hargraves showed us what incredible achievements we might expect from undergraduates. (They would both return to

Dartmouth after earning Ph.D.'s and play decisive roles in the future of computing at Dartmouth.) And I had a high school student, Sidney Marshall, who was taking calculus at Dartmouth. I had him experiment with a language called DOPE (Dartmouth Oversimplified Programming Experiment) on that same LGP-30. DOPE was too primitive to be useful, but it was a precursor of BASIC. (Sidney later became the chief designer for our large time-sharing operating system, which still looks good today!)

KURTZ: John was convinced from the first that a new language was needed, to go along with time-sharing. I felt that there was still a chance of devising subsets of either ALGOL or FORTRAN. In that way, what our students learned would be close to what they would find in the outside world upon graduation. (ALGOL is a now obscure language developed in 1960, used mainly in Europe, and generally considered to be an antecedent of Pascal.)

As an example, I attempted to change the Algol loop statement, which looked like this:

```
for i = 1 step 1 until n do begin . . . ; . . . ; . . . end;
```

One could add comments after the "end," spread things out over several lines, and wind up with this, which is still legal ALGOL:

```
for i = 1 step 1 until n do begin
    . . . ;
    . . . ;
    . . .
end next i;
```

and the code would look more like the language we wanted. But there was no way of achieving further simplification without drastically changing the language.

Similar attempts with FORTRAN also failed; there were just too many ugly features that we wanted to change or eliminate. For instance, the rules as to where commas were needed were hard to remember, the DO loop was always executed at least once, and variable names beginning with I, J, K, L, M, or N were automatically of integer type. (Integer variables in FORTRAN are similar to variables with names like I% in some

current BASICs.) If we had corrected FORTRAN's ugly features, we wouldn't have FORTRAN anymore. I reluctantly had to agree with John that, yes, a new language was needed.

Nonetheless, BASIC was strongly influenced by FORTRAN and ALGOL. For instance, the BASIC FOR-NEXT loop is a direct descendent of both the ALGOL and FORTRAN loops. From ALGOL we borrowed the use of English words, changing "until" to "to," and the feature that the loop is carried out zero times when n (see above) is zero. From FORTRAN we borrowed the feature that the step size is the last element of the DO statement, and can be omitted when the step size is 1.

DESIGN OF A LANGUAGE

From our experiments and from observing a generation of students, we had very definite ideas of what features our language should have:

1. It should be easy to learn for the beginner.
2. It should be a general-purpose language, allowing the writing of any program.
3. Advanced features had to be added so that, if there was a price, it was paid by the expert, not the novice.
4. It should take full advantage of the fact that the user could interact with the computer.
5. It should give error messages that were clear and friendly to the user.
6. It should give fast response for small programs.
7. No understanding of the hardware should be necessary.
8. It should shield the user from the operating system.

We designed the language together. Much of the design was dictated by these eight principles. Where there were choices, we would

argue at length about the "right" way of doing things. Many of the good features of the language were due to this careful design, others to lucky guesses.

We will describe how these principles led us to the original BASIC. Principles 1–3 will be discussed in this section, 4–6 in the next, and 7 and 8 in the section "Computer Independence."

The first principle was always foremost in our minds. Each line should contain one instruction. Each instruction started with a command, and the fourteen commands of the original version were sufficient for elementary programming. We would start lectures on BASIC with an example such as

```
10 LET X = 5
20 LET Y = 7
30 LET Z = X + Y
40 PRINT Z
50 END
```

Most students could figure out for themselves what the program did. We introduced students to the elementary commands in three lectures, then expected them to write programs entirely on their own. When we ran a course evaluation, the main criticism was that the third lecture was unnecessary. So we now give a two-lecture introduction to programming in BASIC.

Principles 2 and 3 deal with additions of more advanced features. BASIC can deal with verbal information as well as with numbers. It handles arrays (lists and tables). It can read and write files. Although the elementary instructions allow one to write any program, one needs more advanced commands for the convenient writing of large programs.

Most languages are spoiled when advanced features are added. To make it a general-purpose language—that is, to achieve principle 2—principle 3 is violated. The designers of such languages are experienced programmers who tend to forget how hard it was to learn their first computer language. They replace elementary tools with sophisticated ones, forgetting that beginners would like to learn the simplest ideas first.

Advanced features were added to BASIC in the following years, usually by student programmers. The main role we played was to insist on a simple, "clean" design and make sure that life was not made harder for the novice. Repeatedly we forced redesign, vetoing new features that made life "just a little harder" for the beginner. We always found that it was possible to add a new feature for the expert without forcing it on the beginner. In a sense the language was built in levels, so that you did not

need to learn about the next, more advanced level until you were ready for it.

Let us use an example from FORTRAN. It has a very fancy capability for formatting numerical output. You can specify one of many different formats, the number of decimal places, and the precise placement of numbers on the line. Unfortunately, the result is a fairly complicated format instruction that we can never remember without looking in the manual. The simple fact is that most of the time you want to see a number, or a few numbers, and you don't care how they are printed. FORTRAN forced you to use a complicated format command even when you didn't care.

In BASIC a printing format is chosen for you. An integer comes out as an integer. A decimal fraction is shown to six places (good enough for almost all applications). And scientific notation is used only when neither of the above is possible. You can specify your own format, but you don't have to. This is typical of BASIC's design: Give the user a solution that is satisfactory most of the time. If the expert needs something fancier, let the expert do the extra work!

Technically this is known as a "default option." We built such defaults into the language wherever possible.

INTERACTIVE COMPUTING (PRINCIPLES 4–6)

Previous languages had been designed for batch processing. The fact that BASIC was designed for interactive computing allowed us to make the language much "friendlier." In batch processing, some of the conveniences built into BASIC would not have been possible. The best example is the use of line numbers for editing (correcting) a program.

Our users were not going to punch cards. They would use a terminal to enter programs. The terminals were not today's lovely CRT screens, but clunky old model 33 Teletypes. (They operated at a sluggish ten characters per second and printed only capital letters.) But with BASIC, if you discovered an error on a line, you only had to retype the line correctly. And you could add or delete a line without having to retype the entire program. All this was achieved by the decision to start each line with a line number.

To correct a line you retyped it, using the same line number. The system automatically replaced the old line with the new one. In our first manual we suggested using multiples of ten for line numbers, so that if

you discovered a line missing, say between lines 40 and 50, you could add it with an appropriate new line number, say 43. Typing the number of a line with nothing after it was the signal for deleting an unwanted line. Beginners very quickly learned how to type corrections, and even we were surprised at how much one could achieve in a single session at a terminal. Until the appearance of modern screen editors, the line-numbering system we devised was the easiest editing system available.

But the most radical use of the interaction possible in time-sharing was the INPUT command (added somewhat later). All languages had a command that functioned like PRINT. It is the "tell me" command. Without it you could never obtain answers. INPUT is the "ask me" command. The running of your program could be interrupted long enough to let you supply a piece of information or to answer a question. This is impossible in batch processing systems, partly because it would be heresy ever to interrupt the computer, but more importantly because it has no way of communicating with the user while the program is running. That single command made possible all truly interactive programs, ranging from management information systems to computer games.

Fast response and good error messages were an important part of the interactive correction of programs. We designed BASIC to give a short message identifying the nature of the problem, and supplied the number of the line on which it occurred. The user could then retype the line or add other lines to correct the problem. We also decided that BASIC should print only a small number of error messages at a time. A list of thirty errors would discourage anyone! And the chances are that later errors were indirectly caused by the early ones. Correcting the early errors and then rerunning the program often were all that was needed. In any case, the user could type RUN again and find out what other errors still existed.

The key to the success of all this was principle 6, fast response. This did not necessarily mean that programs had to run fast. The fact is that most of the time programs do not run at all! Instead they result in one or more error messages.

Therefore, the key to fast response is a program that quickly attempts to translate your BASIC program into machine language, and in the process discovers anything that's illegal. Such a program is known as a compiler. BASIC was designed to allow the writing of a fast compiler. One instruction per line and each line starting with a command word not only make the language easy to learn, but also allow fast compiling. And certain restrictions were imposed to speed up the translation process even more.

We make a major point of this because the implementations of

BASIC known to most users in the world are so-called interpreters. They are easier to write than compilers but have many shortcomings. They are slow in that if a line is carried out one thousand times, an interpreter must translate it one thousand times. That slows up execution significantly even on a modern computer. And since it looks at only one line at a time, its error messages cannot be precise. Errors may depend on the structure of a long segment, not on a single line. Many of the implementations of BASIC on personal computers allow you to run illegal programs! And even if they do spot errors, the program might run for ten minutes before the interpreter reaches the illegal instruction.

Obviously, fast analysis of errors has been written out of most other versions of BASIC. And for a student learning to program, fast response is essential for a sense of accomplishment.

The very first implementation of BASIC in 1964 was a compiler, and all later implementations at Dartmouth have been compilers, including the powerful 1983 version called Dartmouth Structured BASIC®. That may explain why we have a very different feeling about the language than those who judge it solely through some of the miserable implementations on personal computers.

COMPUTER INDEPENDENCE (PRINCIPLES 7 AND 8)

Not having to understand the hardware was a controversial decision. Many programming courses introduced students to the mysteries of hardware before teaching them a language. The result was that most liberal arts students were afraid to take a computer programming course! Considering the vast changes that have occurred in computer engineering in the past decade and the fantastic rate at which new equipment is introduced today, one wonders how much lasting effect those courses had.

And peculiarities of the hardware were built into languages. In 1963 computers could do very fast computations with integers. Noninteger numbers were carried in scientific notation, such as

$$1.239 * 10^{-7}$$

Arithmetic with such "floating point numbers" was much slower. So FORTRAN had two kinds of numbers. We decided that for beginners a number is a number is a number. There is only one kind of number in BASIC. If there is a problem of conversion, it is up to the implementer of

the language to worry about it—not the user. In the first implementation, we carried every number as a floating point number, paying the price of slower calculations but gaining great simplicity for the programmer.

Within a few years manufacturers handled floating point operations by special hardware. This made BASIC faster than other languages—since the new hardware was fast and we didn't have the problem of conversion. Any reason for two kinds of numbers in the language had disappeared.

KEMENY: History repeats itself. When I got an IBM PC, I discovered that its BASIC was spoiled by similar kowtowing to the hardware. But now there are *four* kinds of real numbers! The implementation is ugly, and will be silly as soon as better hardware is available—which will be very soon. (We will have more to say about this in chapter 4.)

Finally, we designed a very friendly environment in which BASIC is used. Simple commands, such as NEW, OLD, SAVE, LIST, RUN, have now become widely accepted. The user can save a program without having any idea of the nature of the file system. This was most fortunate, since we have changed our file devices several times, but users of BASIC had no need to be aware of the change, no matter how drastic. In the Dartmouth implementation these commands are not even part of BASIC; they are part of a monitor, a portion of the operating system that forms a kind of friendly umbrella to protect BASIC and other languages. But most of our users think of the commands as part of BASIC. And that is fine too.

COMPUTING SHOULD BE FREELY AVAILABLE

KURTZ: I always insisted that computing be open-access, like an open stack library. This policy didn't have anything to do with BASIC, but was part of the overall strategy for making computing easy and accessible to all. The alternative, as practiced at most other universities, required applying (in writing) for computing time. If one had a serious application, one might be willing to go through this red tape. But for a

small, spur-of-the-moment problem, one might lose interest in it before the application was filled out and approved.

In the years following, I had continuous arguments with some of my professional colleagues outside Dartmouth about the proper way to "charge" for student computer use. I would be almost the only one arguing that computing costs should be part of tuition, just like library costs. My point was that computing was a fixed cost—it cost the same whether it was used or not—so why not let the students use it? My colleagues would point out that computing was expensive (they were right), was an economic good (which was true), and therefore should be charged for (like movies or hamburgers). I never won that argument, but we did it "our way" at Dartmouth. That particular discussion is now moot. Student computing will soon be supplied almost exclusively by personal computers. Someone will have to pay, of course, but you pay for the access rights, and not for how much you actually use.

KEMENY: In later years I would find Tom's analogy of a library to be a forceful argument. When alumni asked me why we "gave away" computing, I asked them whether they would charge a student for taking a book out of the library. (I confessed that on occasion I was tempted to pay a student to take a book out of the library.)

BASIC IS BORN

KEMENY: After the design was complete, we divided up the task of implementation. Tom supervised the students who were creating the time-sharing system, and I wrote the first BASIC compiler.

Compilers translate programs written in a language like BASIC into machine language, so that the computer can run them. In effect, the compiler teaches the computer how to talk BASIC. Today there is a vast literature on the art of compiler writing, and we routinely teach it to computer science majors. But in 1963 I had not read anything about compilers. Indeed, I had never seen a compiler!

The result was interesting. As a logician I thought that I knew how a compiler should work. And I did invent several very good ideas that made the compiler fast. For example, the limitation on variable names allowed preallocation of memory, and forward transfers were handled by a "transfer table."

There were also some terrible inefficiencies. Students would kid me for years about how I translated formulas. I believe it was the slowest possible method. But, fortunately, a typical program has very few long formulas in it, so the compiler was still very fast overall.

There were also two errors in the resulting compiler.

It calculated $-x^2$ as if it were $(-x)^2$. Since the formula for a normal (bell-shaped) curve involves just such a construct, probabilists and statisticians obtained some amazing results! But a student soon fixed that error. I also used the wrong convention for the function INT (integer part). All in all, having only two implementation errors in the first compiler you ever write is not too bad!

The most frustrating part of writing the BASIC compiler was that I could not afford to wait until our equipment arrived. GE kindly allowed me to do debugging on a computer in Boston. Since they did not have time-sharing (no one did!), they even allowed me to take over the entire computer for brief periods. It would turn out to be the last time I ever had to work on a batch processing system.

Our equipment arrived in February 1964 and was fully functional by March 1. We put in very long days and even longer nights designing, coding, and debugging. There were times when we thought we could never put it all together.

Then, on May 1, 1964, at four in the morning, John Kemeny and a student programmer were sitting at neighboring terminals and typing simple programs in BASIC. They typed RUN at the same time and were deliriously happy when both got correct answers. Time-sharing and BASIC were born.

KURTZ: Not being a night person, I was probably home in bed at that hour, so I missed all the fun.

While John wrote the first BASIC compiler, our students did get a chance to strut their stuff. One of them wrote a compiler for ALGOL that summer, building on the work done earlier on the LGP-30, but that language failed to make its mark. Others eventually took over the maintenance and further development of the BASIC compiler.

Dartmouth received a great deal of praise for the invention of BASIC. But most people missed the fact that the revolution, interactive computing, was made possible by a combination of BASIC and time-sharing. By pulling off a double revolution, the college received only half the credit it deserved!

2
BASIC GROWS UP
1964–1971

Although the development of BASIC worldwide has suffered from having "too many cooks," at Dartmouth the persons concerned with refining BASIC, and there have been many, kept the original principles in mind. That doesn't mean we didn't make changes in the language. It does mean that we tested all changes against those principles. Bringing up BASIC has thus received constant and continuous attention since its birth.

In this chapter we will recount the story of BASIC at Dartmouth from its birth (1964) to its adulthood (1971). This period of rapid growth of BASIC predated personal computers. And yet personal computer versions of BASIC adopted practically none of what we had learned and included in Dartmouth's BASIC before 1971.

In the commercial world, the great god of "upward compatibility" has dictated that bad design choices made early on are not corrected later, because customers have begun to depend on them. We have never had to struggle with such dogmatic principles. Unfortunate bad decisions in early versions were "corrected" in later versions. Being an educational institution, we then simply "reeducated" our students and colleagues.

This course hasn't always been easy. Reeducation is a hard job. New manuals have to be written and seminars given. Even so, in recent times we have found ourselves in the company of several vintage versions of Dartmouth BASIC that, like old soldiers, never die but seem to keep hanging around. It is turning out to be a major task to come back to a single version of BASIC. If it is hard for us, think of how hard it must be in the outside world!

During this period, BASIC at Dartmouth made the transition from a language suitable only for small programs to a language suitable for building large applications. The date 1971 is important for two reasons. First, before that date the changes to BASIC came thick and fast; hardly a year went by without some major improvement. Around 1971 it became apparent that stability in a widely used language was more important than new features. No longer could changes be made without considering the opinions of others and taking into account the consequences, well meaning though our intentions might have been. In response, we brought out BASIC the Sixth (the sixth edition). It was particularly stable and became the workhorse language at Dartmouth for more than a decade.

Second, that 1971 date marked a watershed in that time-sharing as a way of distributing computing resources was approaching its pinnacle, whereas the major developments of interactive graphics, personal computers, and structured programming still lay in the future. The story of BASIC after 1971 will be continued in chapter 3.

A note about the use of "we" in this chapter. The authors had sole responsibility for BASIC the First. As time went on, others became involved. At times, when we were busy with other tasks, we just made suggestions. More recently, we both became more deeply involved in the progress of BASIC. Others continue to be involved as well. So in the following paragraphs, "we" refers to all the folks involved, and not just the two of us

FIRST BEGINNINGS

The first version of BASIC at Dartmouth, vintage 1964, was virtually identical in capability with many of the versions now found on the smallest personal computers. The ANSI standard for Minimal BASIC (which came out in 1978 in the United States)[1] is also very close to that 1964 Dartmouth version.

The 1964 language contained these statements: LET, PRINT, END, FOR, NEXT, GOTO, IF THEN, READ, DATA, DIM, DEF, GOSUB, RETURN, and REM.[2] The GOTO, IF THEN, and GOSUB referred to line numbers only. Only numeric constants and variables were included. DEF allowed a limited number of single-line function definitions, those with names FNA through FNZ. GOSUB and RETURN provided a primitive form of internal subroutine. The stock of variable names was limited to single letters or single letters followed by single digits. Array names were limited to single letters. In the absence of a

DIM statement, arrays were declared by default according to their first appearance in the program, either as a vector or as a matrix (list or table), and with default dimensions of 1 to 10.

In the rest of this chapter we outline the growth, from those simple beginnings, of several parts of BASIC, such as input-output and strings. But first we shall discuss the association between Dartmouth and General Electric from 1964 until 1972, an important topic because GE helped spread BASIC to the outside world.

THE DARTMOUTH–GENERAL ELECTRIC CONNECTION

There were two main genealogical lines growing out of the original BASIC at Dartmouth. Large-machine BASICs in later years were probably direct descendants of GE BASIC, which was in turn a direct descendant of Dartmouth BASIC. On the other hand, most BASICs on small machines were descendants of versions that first appeared on the Hewlett-Packard HP-2000 or the Digital Equipment PDP-8, which makes them more like first cousins, once removed. The HP-2000 BASIC was originally based on GE BASIC, but we note later that GE BASIC and Dartmouth BASIC were virtually one and the same until around 1970.

As we described in chapter 1, Dartmouth built its several time-sharing systems around hardware manufactured by General Electric. Less well known are the details of how it all happened and what the results were.

GE hardware was selected because it was the only major brand that offered what we wanted at a reasonable price. Even this was partly a fluke, for the following reason: our design for a time-sharing system required two computers, one to do the "computing" and one to "communicate" with the time-sharing terminals. GE had both kinds of computers to sell, to be sure. The fluke was that the communications computer had been designed, not for time-sharing, but for a Teletype switching network of a major automobile maker. Either the other vendors did not have a communications computer for sale or, if they did, it was far too expensive.[3]

The two GE computers were the GE-235 and the Datanet-30. Later, GE attached the name GE-265 to the combination of the two computers and the Dartmouth time-sharing software. At one time, as many as

seventy-five such systems were in use around the United States.

Although GE offered us ample educational discounts, as did some of the other vendors, the discounts for our project were not much larger than for GE's other educational customers. Nor were they in "payment" for the software we were about to build. (The "payment" came later.)

As we reported in chapter 1, the time-sharing system, together with BASIC, first saw the light of day on May 1, 1964, at 4:00 A.M. That was when two BASIC programs were first run simultaneously, both giving the correct answers. By June 1964, the two original Teletype model 33 terminals had grown to eleven, and by that fall, to twenty, most of them being Teletype model 35s. We observe with pride that most of these were for student use; the needs of the faculty and the administration came later.

GE installed a copy of the Dartmouth system in its plant in Phoenix, Arizona, in December 1964. Shortly thereafter they started selling time-sharing services on it. This was the start of what later became the largest commercial time-sharing service bureau in the world.

That first GE system was identical with the original Dartmouth system, even down to the BASIC. Once weaned from Dartmouth, GE BASIC grew in a different direction to serve the needs of its customers.

The relation with GE was cooperation between industry and a university at its best. But there was occasional friction, particularly with our student programmers. We will recount one such incident.

Some of our students were hired by GE during vacation time to help bring up a commercial version of our time-sharing system. We knew that our first system was not very secure. But GE felt more security was needed in a commercial environment, and they set about to achieve it. One day several GE employees told one of the students that their system was now completely safe from tampering. Our students waited until a senior vice-president came on an inspection tour. That day, when a terminal was turned on, it typed not "GE Time Sharing," but "The Jolly Green Giant strides through the Valley of the Giants!" (Green is the Dartmouth school color, and the Valley of the Giants is near the former GE computer plant in Phoenix, Arizona.)

Any professor could have told GE that you never tell bright college students that there is something they *cannot* do—unless you want to be sure that they will do it. This has got to be one of the earliest examples of the art of computer hacking.

In late 1965, GE offered Dartmouth a deal that we could not afford to turn down. They offered to place fancy new hardware at Dartmouth if we would join them to develop a time-sharing system, including a

BASIC, for the new machine. (This offer was the "payment" mentioned earlier.) This project was officially launched on September 30, 1966. The fancy new hardware, a complete GE-635 computer, arrived in December 1966; additional major components arrived three years later. Dartmouth people did write the BASIC compiler, and several other parts of the system. GE people wrote the time-sharing operating system.

The operating system that GE wrote was supposed to be an interim one. It was called Phase I. Once done with the BASIC compiler, Dartmouth was to write a more sophisticated version. This Dartmouth did, finally achieving it in the spring of 1969. It was called Phase II. By that time GE had decided to stick with Phase I. We suppose it was because they had grown accustomed to it in the intervening year and a half. They did rename their new child Mark II. (By this time, the original system based on the GE-265 had been renamed Mark I.) Back at Dartmouth, we rechristened our offspring DTSS (for Dartmouth Time Sharing System) since "Phase II" sounded like the name of a bath soap. Just as they had in 1965, the versions of BASIC on the Dartmouth campus and on the GE network again began to diverge.

GE continued its relationship with Dartmouth until September 30, 1972, when interest in joint projects waned. The joint effort left GE with the largest commercial time-sharing network in the world. Dartmouth ended up with hardware far more expensive than it could have afforded on its own.

When GE and Dartmouth started on separate paths around 1969, GE took with it the version of BASIC known as the Fifth Edition. That frankly inferior version lasted at Dartmouth only about a year and was replaced by BASIC the Sixth in September of 1971. What this meant for the geneology of BASIC was that the new line of BASICs outside of Dartmouth was spawned from genetically inferior stock.

THE GROWTH OF THE PRINT STATEMENT

The original 1964 BASIC had the simple input and output statements READ, DATA, and PRINT. READ and DATA worked then much as they do today. PRINT was more primitive than present-day PRINT statements, however.

The first PRINT statement required commas between the items to be printed. Each comma meant "go to the next print zone," which was fifteen characters wide. Crude tabular output could thus be achieved

without special formatting. Labels (strings of characters within quotation marks) could also be printed. This earliest BASIC could process strings, but did allow quoted strings in PRINT statements. An example is the following:

```
100 PRINT "Answer is", X
```

Within six months, we realized we sometimes wanted to "pack" the output. That is, we might want to print numbers more closely spaced than the comma-induced print zones allowed. So we added the semicolon (;) to separate items in a PRINT statement, as in

```
100 PRINT X; Y; Z
```

One sometimes gets lazy. We did with the semicolon. Our machine had twenty-bit words, which held three six-bit characters. (This was before today's eight-bit bytes.) We lazily coded the ; to have the print mechanism move to the next multiple of three character positions so that we would always be printing a full word's worth of characters. Thus, in "packed" output, there might be either one, two, or three spaces between numbers, depending on where the boundaries of the machine words fell. Thus,

```
100 PRINT 1; 10; 100; 1000
```

would thus produce something like

```
1  10    100   1000
```

rather than

```
1  10  100  1000
```

We quickly corrected this "error." But it is interesting that one famous version of BASIC, CALL-360 BASIC for the IBM-360, patterned on our BASIC with our permission, retained this feature long after we had discarded it.

The PRINT statement with the semicolon allowed you to print numbers or strings in just about any format you wanted. It was a lot of work, but you could print a number in a certain format by printing bits and pieces of it with the ; as a separator. To make this task easier, we even devised a special function, the STR$ function, to convert a number to a string, but without the leading and trailing spaces.

It wasn't until around 1971 that we introduced a fancier formatting tool, the PRINT USING statement. A similar feature is present in many modern versions of BASIC. Its main purpose was to allow decimal points to line up when printing financial amounts. As an example,

```
PRINT USING "The balance is -##,###.##", B1
```

would cause the dollar amounts to be printed with the decimal points aligned, regardless of the size of the balance B1. (In Standard BASIC, a colon is used in place of the last comma.)

THE GROWTH OF THE INPUT STATEMENT

The most surprising fact about the INPUT statement is that it did not appear until almost two years after the birth of BASIC. There were two reasons. One was technical. Our first time-sharing system had two separate computers. The front-end computer handled all editing and carried out all commands (such as NEW, SAVE, etc.). The rear-end computer compiled and ran the programs, more or less independently of the front-end computer. To implement the INPUT statement, we had to arrange for a running program in the rear-end computer to pause, signal the front-end computer that input was needed, and wait for that input. While this was going on, the rear-end computer would be waiting and doing nothing. This technical problem was resolved by "swapping" to the disk any program waiting for input, and running other programs, until the input for the swapped program was provided. This particular solution is now well known, but it didn't occur to us until late 1965, almost a year and a half after BASIC was born.

The other reason was that we wanted to provide fast turnaround for small jobs, but did not place a high priority on being able to write programs that were in themselves interactive. Our first priority was to be able to "interact" with the computer system in order to create, check out, and run small programs. Only later did we realize the importance of "interacting" with programs, programs that might play a game of checkers or provide language drill and practice.

After we introduced the INPUT statement, another problem surfaced. The INPUT statement assumed that commas separated the numbers or strings being inputted. It was thus impossible to input a string that had a comma. Even today, some personal computer versions of BASIC have no provision for inputting, say, an English sentence that might contain commas. More to the point, such versions of BASIC cannot input lines that are themselves lines from BASIC programs, thus making it impossible to write a program to "prettyprint" another BASIC

program! (Prettyprinting is printing or listing a program in some nice format, such as with indentation to show the structure.)

Realizing this fault, we introduced the LINPUT statement in early 1969. It inputted an entire line from the terminal as is, including all commas and leading and trailing spaces, an ability crucial for text manipulation. The LINPUT statement is now the LINE INPUT statement in Standard BASIC.

MAT STATEMENTS AND THE DEFAULT LOWER BOUND

BASIC the First had only simple arrays of numbers. The arrays were either one-dimensional (called lists or vectors) or two-dimensional (called tables or matrices). If you wanted to tell BASIC exactly how big your lists or tables should be, you used a DIM statement. If you didn't care, you automatically got (by default) lists with subscripts from 1 to 10, or tables with 1 to 10 for both rows and columns. In a DIM statement, you got to specify the upper bound; the lower bound was always 1, by default.

We are now about to start a tale of woe about the default lower bound. Should it be 1, as it was in the beginning, or should it be 0, as we later thought? It took almost twenty years to straighten this one out. The problem was complicated by the arrival of MAT statements, as we shall see.

Most people start counting with 1. But mathematicians and certain others start counting with 0. For instance, the coefficients of a polynomial are usually named c(0) through c(n). In order to use BASIC for calculations on such quantities, BASIC had to allow lists (vectors) to have a zeroth element. That seemed easy enough—we quickly made 0 the default lower bound. Thus, you got to specify the upper bound in a DIM statement, but the lower bound was always 0. But if you used "DIM X(15)," you actually got sixteen elements, those numbered 0, 1, . . . , 15 (count'em). If you didn't use a DIM statement, you got a list with eleven elements, those numbered 0, 1, . . . , 10.

This convention held for a long time. Meanwhile, we added MAT statements. These allow operations on lists or tables in a single statement. For instance, suppose you had a table (or matrix) with R rows and C columns, and you wanted to set all the elements of a table to 0. You could do it either with

```
FOR I = 1 TO R
    FOR J = 1 TO C
        LET T(I,J) = 0
    NEXT J
NEXT I
```

or with

```
MAT T = ZER
```

The next question was, What happens to row (and column) zero? At first, nothing, we thought. Then we decided that MAT statements should operate on the table. So now, when you did

```
MAT T = ZER
```

it meant

```
FOR I = 0 TO R
    FOR J = 0 TO C
        LET T(I,J) = 0
    NEXT J
NEXT I
```

(Each loop starts with 0, not 1.) This caused an awkward problem. Remember that

```
DIM T(10,10)
```

actually creates a table having 121 elements in it. Remember also that the MAT statements used all 121 elements. Suppose one wished to invert a 10-by-10 matrix, a common mathematical task. One would have to use

```
DIM T(9,9)
```

and remember that the numbering of the rows and columns goes from 0 to 9 instead of from 1 to 10. This conflicts with common sense.

The final chapter of this tale was not written until much later. When ANSI Minimal BASIC came out in 1978, its default lower bound was 0, in keeping with most of the early versions of BASIC. But there were those who still felt that the default lower bound should be 1. The BASIC Standard Committee was in such a quandary over this point that they invented a new statement, the OPTION BASE statement. It can specify the lower bound to be either 0 or 1, as in

```
OPTION BASE 0
```

or

```
OPTION BASE 1
```

Technically, this solved all problems, because the programmer could have either one. But there was still a major problem for beginners. They shouldn't have to learn about the OPTION BASE statement, but they would be surprised when they used "DIM X(10)" and got a list with eleven elements in it.

The final solution, not reached until 1983, was to have the default lower bound be 1, which is the commonsense value, and allow those who need the zeroth element to declare it with something like "DIM T(0 to 10)." Dartmouth BASIC had come to this conclusion in 1979, but it took the ANSI committee four more years to see the light. What about MAT statements? Oh yes, MAT statements always operate on the entire array. Most of the time, this means you start numbering the elements from 1. But if you want to use funny ranges, like "DIM T(−5 to 7, 0 to 3)," you can. And if you use MAT statements, you get what you ask for.

CHARACTER STRING PROCESSING

We soon realized that working with strings of characters is as important as working with number. The very earliest versions of BASIC allowed strings in DATA statements, assigning them to string variables and printing them. String constants appeared in quotation marks, as they do today. That was easy. But we needed a way to distinguish a numerical variable from a string variable. We decided to add a dollar sign ($) to distinguish a string variable from a numeric variable.

Why did we choose the dollar sign? There were not so many keys on a Teletype, and we needed to find one that had not yet been used for BASIC. Of the few remaining ones, none seemed very appropriate. Then one of us observed that $ looks like *S* for *string*, and the convention was adopted. Had we known how many millions would end up using "name$" and "word$" in BASIC, we might have given more thought to the choice!

We usually want to do more than just INPUT or PRINT strings. One can add, subtract, multiply, and divide, and compare numbers. What can one do with strings? One can concatenate them (join two shorter strings to get a longer one), cut them into pieces (separate first and last names, for instance), find out if a longer string contains a certain shorter one, and compare them. In comparisons, just as the number 1 is < (less than) the number 2, the string "A" is < (occurs earlier in the alphabet than) the string "B". So far, no problem. But what about string

operations? The symbols + and − have been used for centuries to denote numerical operations. But we weren't sure what symbols to use for string operations. So we took the easy way out—we changed the string of characters into a vector of numbers!

The CHANGE statement placed the ASCII (ASCII stands for American Standard Code for Information Interchange, the character code system most widely used in the United States today) code value for the first character into the first element of a vector, the second into the second, and so on. Once the string was in numerical form, the programmer could perform literally any operation—concatenation, search for particular characters, conversion to uppercase or lowercase, etc. It might be messy, but it could be done. Once the operation was finished, another CHANGE statement would convert the result back to a string. It was not clean, since the programmer had to be aware of the numerical values for the character codes, but it served us well until we had a better idea of what to do.

The following program fragment shows how to use the CHANGE statement to "extract" the fifth and sixth characters of a string to form a new string. (The word *is* will be printed.)

```
100 LET S$ = "Now is the time"
110 CHANGE S$ TO S
120 LET T(1) = S(5)
130 LET T(2) = S(6)
140 LET T(0) = 2
150 CHANGE T TO T$
160 PRINT T$
```

Setting T(0) to 2 told the second CHANGE statement the number of characters in the new string.

The CHANGE statement allowed doing anything with a string, but it certainly wasn't convenient. Most people would rather not convert a string to a list of numbers before doing anything with it. We therefore cautiously added one string operation and several string functions. The one operation, concatenation, was denoted by &. The several string functions were POS, for locating a particular substring in a string; SEG$, for providing a substring; and LEN, for finding its length. SEG$ gave way, in some versions of BASIC, to LEFT$, RIGHT$, and MID$, with MID$ being more or less the same as SEG$.

Extracting the fifth and sixth characters could now be done as follows:

```
100 LET S$ = "Now is the time"
110 LET W$ = SEG$(S$, 5, 6)
120 PRINT W$
```

Later, in Modern BASIC (see chapter 3), extracting the fifth and sixth characters would be done something like this:

```
100 LET string$ = "Now is the time"
110 LET word$ = string$[5:6]
120 PRINT word$
```

This last notation is certainly easier to read.

As another example, let's see what one had to do over the years to convert all the lowercase letters in a string into uppercase letters. That is, if one started with

```
S$ = "Now is the time"
```

one might want

```
T$ = "NOW IS THE TIME"
```

With the CHANGE statement, the code might look like this:

```
200 CHANGE S$ TO T
210 FOR I = 1 TO T(0)
220 IF T(I) < 97 THEN 250
230    IF T(I) > 122 THEN 250
240       LET T(I) = T(I) - 32
250 NEXT I
260 CHANGE T TO T$
```

To understand this code, you have to realize that the ASCII code for *a* is 97, and the code for *z* is 122. You also have to realize that the ASCII code for *A* is 65, which is 97 − 32. Which explains the 32. Got it?

With &, SEG$, and LEN, the code might look like this:

```
200 LET T$ = ""
210 FOR I = 1 TO LEN(S$)
220       LET C$ = SEG$(S$,I,I)
230       IF "a" <= C$ AND C$ <= "z"
240       THEN LET T$ = T$ & CHR$(ASC(C$) - 32)
250       ELSE LET T$ = T$ & C$
260       CONTINUE
270 NEXT I
```

Here, the only magic number is 32, which is the difference between the ASCII codes for the lowercase and uppercase letters. The ASC function (line 240) is the reverse of the CHR$ function. That is, if CHR$(65) is "A", then ASC("A") is 65. (The above example is written in SBASIC, a version of BASIC used at Dartmouth between 1975 and 1980.) Finally, in any modern version of BASIC, the code might be nothing more than

```
200 LET T$ = ucase$(S$)
```

FILES ARE ADDED

By 1969 we had recognized that BASIC was no longer just a simple language for teaching and writing short programs. Any application worth its salt needed to use files.

If your programs are small and operate only on small sets of numbers or strings, DATA statements are perfectly reasonable. But what if there are many numbers or strings to deal with? If you want to change some of the numbers, should you have to change the DATA statements in the program? With extremely large sets of numbers or strings, the size of even simple programs would grow enormously large, because of the number of DATA statements needed.

Things become simpler if we separate the program from the data it operates upon. The data are kept in files and the programs are once again quite short. In some systems, including Dartmouth's, the program itself is kept in a file. This allows a program to be operated on by another program, perhaps to perform some task like prettyprinting.

We introduced two types of files: terminal-format and random-access. Terminal-format files are nowadays called display-format files, character files, or text files. Random-access files were internal-format (that is, not directly listable) and could be either numeric or string, but not both. The type of the file was indicated by, of all things, a trailing character in the file name. Furthermore, the actual names of the files were part of the program; to change those names, you had to change the program. For instance,

```
FILES data%, names$20, explain
```

indicated that file number 1 was a random access numeric file (because of the %), that file number 2 was a random-access string file with 20 as the maximum string length (because of the $), and that file number 3 was terminal-format (because its name "explain" contained neither a % nor a $).

A complete program to print the lines in a terminal-format file is:

```
100 FILES miscdata
110 IF END #1 THEN 150
120    LINPUT #1: A$
130    PRINT A$
140    GO TO 110
150 PRINT "All done."
160 END
```

Line 110 is not an error catcher, as in some current BASICs; rather, END

#1 is "true" if the program is now at the end of the file and further LINPUT statements are not possible.

One problem with that first approach was that you had to change the program (the FILES statement) in order to change the names of the files being accessed. We later added a file opening statement so that the user of the program could supply the name of the file after the program had started up.

```
100 FILES *
105 FILE #1: "miscdata"

(Lines 110-160 as above.)
```

Finally, we realized that the FILES statement served the trivial purpose of declaring that there would be a file number 1, to be named later. The FILES statement was discarded.

```
100 PRINT "Enter file name";
103 INPUT F$
105 FILE #1: F$

(Lines 110-160 as above.)
```

This is essentially the form used today in most versions of BASIC today.

THE GROWTH OF SUBROUTINES

Although one could not write enormous programs in the original BASIC, its authors realized early on that large programs ought to be broken down into small units. The main reason is well known. Some tasks are needed in several places in the program. Rather than duplicate the code each time it is needed, it is better to set up a "subroutine" just once and "call" it each time it is needed.

The original BASIC had only the GOSUB (to a line number) and RETURN for subroutines. Even this facility was provided grudgingly, as the following quotation from the first BASIC manual shows: "A favorite device of programmers is the 'subroutine.' This is a program within the program, which may be called upon one or more times by the main program. It is a luxury item, in that it is always avoidable, but many programmers find it convenient to organize long programs in subroutines."[4]

The simple GOSUB and RETURN were not adequate for large programs. For one thing, you quickly ran out of easily rememberable line numbers. For another, the entire program might not fit into memory. This led to our next experiment, a crude form of subroutine that was really a type of partial overlay.

We called these new entities "subprograms," since they could be programmed separately, each having its own line numbers. Once in a given subprogram, one could jump around with GOTOs and if-thens, but one could leave the subprogram only with a RETURN statement. On the other hand, these subprograms shared the same variables as the main program. Thus, variables were shared but line numbers were not. For those familiar with FORTRAN, this design is close to "chaining with COMMON."

The following example is taken from the 1970 manual.

MAIN program

```
100 SUB INPUT; NEG; POS
110 GOSUB #1
120 GOSUB #N
130 PRINT Y
140 GO TO 110
150 END
```

INPUT subprogram

```
100 INPUT X
110 LET N = (SGN(X)+5)/2
120 RETURN
130 END
```

NEG subprogram

```
100 LET Y = SQR(-X)
110 RETURN
120 END
```

POS subprogram

```
100 LET Y = SQR(X)
110 RETURN
120 END
```

Line 100 in the main program associates the names INPUT, NEG, and POS with the subprogram numbers #1, #2, and #3, respectively.

This small example shows that line numbers were kept apart but that all three subprograms shared the variables X, Y, and N. The subprograms were "invoked" with a GOSUB statement, and RETURN sent them back to the main program, just like the old-fashioned GOSUB

and RETURN. The reader is invited to work through this example, to confirm that it really does work!

This new type of subprogram didn't really catch on, and lasted only about a year. It had several flaws. First, you had to refer to the subprograms by number rather than by name. Second, the subprograms could not have arguments. Third, subprograms could not have a private stock of variable names, but had to share all variables with the main program. Nonetheless, this form of program segmentation allowed us to write quite large applications. One of the largest was the code for Project IMPRESS, a statistical system for social science data analysis, designed especially for educational use and used by Dartmouth and many other institutions.[5]

Allowing subroutines to have arguments is extremely important. With only the GOSUB and RETURN statements, one has to use regular variable names to communicate with the subroutine. As an example, suppose the subroutine simply adds two numbers. We might program it as follows:

```
100 INPUT a, b
110 LET x = a
120 LET y = b
130 GOSUB 1000
140 LET c = z
150 PRINT c
160
1000 REM Subroutine—adds two numbers
1010 LET z = x + y
1020 RETURN
1030 END
```

Of course, one would not program such a simple application this way. But suppose that the subroutine is used at several points in the main program. The LET statements just before and just after the GOSUB statement then become essential. This construction is obviously awkward, as any who have tried to use it will realize.

By the time BASIC the Sixth came into being, we had realized that we needed genuine external subroutines. So we added them to BASIC. We still called them subprograms, but we retired the GOSUB overlaying statement. Instead, the subprograms were "called" with a CALL statement, as is the case in most languages today. They eliminated the flaws of the earlier GOSUB-type subprograms—they were called by their name, they allowed arguments, and their internal variables were distinct from those in the main program. A typical example in this 1971 version of BASIC is

```
100 INPUT a, b
110 CALL "SUM": a, b, c
120 PRINT c
130 END
140
150 SUB "SUM": x, y, z
160     LET z = x + y
170 SUBEND
```

We were not sure whether we might someday want to use something like

```
105 LET A$ = "SUM"
110 CALL A$: a, b, c
```

so we paved the way by having the names be quoted strings. This was later changed to the now-standard approach of having the subprogram name appear unquoted. A colon (:) preceded the argument list, which is also different from the now-standard approach of using parentheses to surround the arguments. Looking ahead, the same example in Standard BASIC (see chapter 3) appears as

```
100 INPUT a, b
110 CALL sum (a, b, c)
120 PRINT c
130 END
140
150 SUB sum (x, y, z)
160     LET z = x + y
170 END SUB
```

THE USE OF INDENTATION

Programmers now generally agree that indentation, properly used, helps them understand programs. We usually indent the inside part of FOR-NEXT loops, etc. One of our gripes about some current microcomputer verions of BASIC is that they don't allow indentation at all. In contrast, we recognized its importance as early as 1965, as the following dated excerpt shows.

```
SALES1   10:49   20 DEC. 1965

10  FOR I = 1 TO 3
20      READ P(I)
```

```
30  NEXT I
40  FOR I = 1 TO 3
50      FOR J = 1 TO 5
60          READ S(I,J)
70      NEXT J
80  NEXT I
90  FOR J = 1 TO 5
100     LET S = 0
110     FOR I = 1 TO 3
120         LET S = S + P(I) * S(I,J)
130     NEXT I
140     PRINT "TOTAL SALES FOR SALESMAN" J, "$" S
150 NEXT J
900 DATA 1.25, 4.30, 2.50
910 DATA 40, 20, 37, 29, 42
920 DATA 10, 16, 3, 21, 8
930 DATA 35, 47, 29, 16, 33
999 END
```

Indentation clearly reveals the nature of the loops and double loops. We might criticize the program on other counts: we don't approve of the "jog" between two-digit and three-digit line numbers; there are no comments; and the program uses uppercase letters only—but our terminals were model 33 or 35 Teletypes, which had only uppercase letters. And the later addition of MAT statements makes all those loops unnecessary.

A recent scientific study has confirmed that indentation, properly done, can help one read and understand a program.[6] But we knew that all along!

AT LAST, A STABLE BASIC

After six hectic years of rapidly changing versions of BASIC, punctuated by frequent correcting of earlier errors, it finally dawned on us, as we explained earlier, that a new version was needed, one that was actually planned and designed. Planning occurred during the spring and summer of 1970, coding and testing during the 1970–71 school year. The summer of 1971 was spent in what is now sometimes called "beta testing" (testing by selected outside users) and in converting libraries of programs. Switchover to the new version occurred in September of 1971 with no ill effects. A tribute to the thought and care that went into its planning and development is that it served as the principal version of BASIC for Dartmouth until 1979, and as a subsidiary version even into the 1980s.

Every effort was made to have it run very fast, to permit large numbers of students simultaneously to code, compile, and test in the time-sharing system. It included an on-line debugger. It also allowed large programs by automatically overlaying subprograms no longer needed with ones that were needed. And these subprograms could call themselves recursively. All in all, BASIC the Sixth was a huge success. With it, we were ready for the seventies.

3
BASIC MATURES 1971–PRESENT

As we entered the decade of the seventies, there was scant warning about the computing revolutions brewing. We had settled back to enjoy our new version of BASIC the Sixth. Little did we realize the impact that fancy graphics and structured programming would have by the end of the decade. And there was little hint that within a dozen years we would have computers sitting on our desktops that would be as powerful as the huge central time-sharing system that served us so well.

In this chapter we will describe the revolutions in plotting, structured programming, and personal computers, and how they affected the growth of BASIC. We will also describe briefly the efforts to develop an official standard for BASIC during this period.

The first revolution to arrive was plotting. At the end of the sixties, there were few CRTs or video displays capable of plotting curves. (CRT is short for cathode ray tube, which is just a fancy name for a television picture tube.) Except on expensive research computers, the only alternatives were small flatbed plotters in which specially mounted ink pens or nylon-tipped pens roamed over the bed, tracing pictures as they went. For each new picture, one had to lift up the pen and replace the sheet of paper. The plotters did not have keyboards, and so needed an extra output pass to plot the results. In short, they could not be used interactively. This was shortly to change.

About the time BASIC the Sixth arrived, the science of structured programming was beginning to take hold. It is fair to credit the famous computer scientist Edsgar Dijkstra with getting the ball rolling. In 1965

he delivered a paper in which he pointed out that the difficulty of testing and modifying a program seemed to be proportional to the number of GOTO statements it contained. When he published his now-famous short letter entitled "Goto Statement Considered Harmful," his views could no longer be ignored.[1]

We said above that we had no hint of the personal computer revolution about to hit us. Actually, we did have a hint. Some of us had heard Alan Kay's presentation at the ACM conference in 1972, in which he described the technical feasibility of a personal computer the size of a notebook, having the power of a large computer, and costing $100.[2] The only part of his prediction that is not yet common is the flat display—most of us still have to struggle with bulky CRTs.

The seventies also saw the beginnings of a Standard BASIC. Formed in 1974, the X3J2 Committee (don't they pick quaint names for these committees!) two years later produced a small standard for Minimal BASIC, which received official approval in 1978. X3J2 then started on full BASIC. But by then it had to cope with a moving target—the explosion in personal computers, structured programming, and graphics—and it took far longer than two years to devise the standard for the full language.

In the remaining parts of this chapter, we refer to "Modern BASIC" and "Standard BASIC." By "Modern BASIC" we mean any version of BASIC equipped with the tools of structured programming and graphics. By "Standard BASIC" we mean any version of BASIC that adheres to the draft proposed (at the time of this writing) American National Standard (dpANS) for BASIC.

PLOTTING

Interactive plotting was the first innovation of the seventies to hit us. It arrived in early 1968 and therefore predated personal computers. In fact, it even predated CRT plotting terminals. One of our colleagues, Arthur Luehrmann (now with Computer Literacy, Inc., in Berkeley, California), devised a way to attach small flatbed plotters to ordinary Teletype terminals. Programs could then send both text messages and graphic output to the same terminal. The distinction as to whether the Teletype terminal printed the message or sent it to the plotter for graphing was based on certain control character sequences embedded in the message.

Before this time (and except on fancy research computers), one

could prepare graphical output, but it had to be sent to a separate machine, so one could not examine the picture until much later. With the new approach, users could interact with their programs as the pictures were drawn, making changes and corrections when they didn't like what they saw. The big step forward was the "interactive" part. It was now practical to have students plot curves from physics, say, as a routine homework assignment.

Professor Luehrmann realized that most users did not wish to deal with the special character sequences for driving the plotting devices, and in particular would not care to recast their own problem into the coordinate system of the plotter. He therefore wrote a series of BASIC subprograms (which were invoked with GOSUB statements; see chapter 2) for carrying out the graphical operations. These subprograms were collected together into a "library."

The most important subprogram allowed the user to specify his or her own coordinate system. The subprogram would then "convert" the points and lines in this coordinate system to raw plotter coordinates. An example will illustrate what we mean. The plotting surface is usually divided into a certain number of points in one direction and another number of points in the other direction. Thus, the x-coordinate might run from 0 to 767, while the y-coordinate might run from 0 to 511. Any line or point drawn on the surface must be expressed in terms of these coordinates.

If the user wished to plot the sine curve from 0 to 2*pi, he would have to write code to associate an x-value of 0 with 0, an x-value of 2*pi with 767, a y-value of −1 with 0, and a y-value of +1 with 511 on the plotter. Then where would the point x = .1, y = sin(.1) = .09883 lie? A little calculation shows that the point (.1, .09883) in the user's coordinates lies at (12, 281) on the screen. These simple (for the computer) calculations, different for each plotter, are sometimes called the "window transformation." The subprogram that carried them out was thus the heart of Professor Luehrmann's subprogram library.

Besides the "window transformation" subprogram, Professor Luehrmann's library also contained all the subprograms needed for drawing the actual points. A single library statement thus brought in all the correct subprograms for each type of plotter. All other statements in the user's program would be the same, including the subprogram calls themselves! Thus, the program and the resulting plotted output would be virtually identical, regardless of the actual plotting device used. The only difference was the "library" statement near the start of the program. To change to a different type of plotter, only this statement had to be changed.

When BASIC the Sixth appeared in 1970, Professor Luehrmann

converted his old-style subprograms to fit the new language. A complete program to plot the sine curve is shown below.

```
100 LIBRARY "PLOTLIB***:TDI"
110 DIM O(150)
120 READ X0, X1, Y0, Y1
130 DATA 0, 6.29, -1, 1
140 CALL "LIMITS": O(), X0, X1, Y0, Y1
150 FOR X = X0 TO X1 STEP (X1 - X0)/40
160     CALL "LINE": O(), X, SIN(X)
170 NEXT X
180 CALL "FINISH": O()
190 END
```

The LIBRARY statement in line 100 served to name the specific file containing all the graphics subroutines for the desired plotter. Nowhere does the programmer have to know the details of the TDI plotter, especially its own internal coordinate system. If a Tektronix 4010 terminal were used instead, all one had to do was to change line 100 to

```
100 LIBRARY "PLOTLIB***:TEK4010"
```

The special vector O(), contained all information that had to be shared among the plotting subroutines. The subroutine "LIMITS" in line 140 established the "window," that is, the coordinates desired by the user. The subroutine "LINE" drew a line from the current position to the new position given. Certain necessary cleanup operations were performed at the end by the subroutine "FINISH".

Professor Luehrmann soon realized that plotting would be even more convenient if the plotting could be done with statements in the language itself, rather than through subroutine calls. He therefore designed a preprocessor called PLOT to allow this. Dartmouth students could now use PLOT statements instead of issuing CALLs to subroutines. Plotting thus became (in 1974) an integral part of Dartmouth BASIC.

As an example, the above program could then be rewritten

```
100 PLOTTER "TDI"
110 READ X0, X1, Y0, Y1
120 DATA 0, 6.29, -1, 1
130 WINDOW X0, X1, Y0, Y1
140 FOR X = X0 TO X1 STEP (X1 - X0)/40
150     PLOT X, SIN(X);
160 NEXT X
170 END
```

Gone are the subroutine calls. Gone also is the call to FINISH. One major difficulty remained, however. The PLOTTER, WINDOW, and PLOT statements were converted to CALL statements by a preprocessor to allow the resulting program to run under regular BASIC. In addition, the preprocessor had to establish the vector O() by secretly adding a DIM statement for it. The preprocessor also established the matrices P(,) and Q(,), in case transformations were used. The programmer therefore had to avoid using these three arrays. Strange behavior occurred when the programmer, through ignorance or error, used one or more of these special arrays. And this happened often, since in those days programmers had only twenty-six array names to play around with—the single letters *A* through *Z*.

The preemption of the special arrays can be eliminated if the graphics capability is included in the language itself, rather than being provided through a preprocessor. (This improvement was made in 1979.) This experience also taught us a lesson about languages: subroutines need to be able to share variables and arrays that are otherwise completely hidden from the rest of the program. For plotting, we want to share the arrays O(), P(,), and Q(,) between the graphics subprograms, but to have them hidden from the user, who can then use the same arrays for his or her own purposes. This feature, sometimes called "modules," was added to Dartmouth BASIC in 1979, but has not yet appeared in any other version of BASIC.

STRUCTURED PROGRAMMING

In chapter 5 we will describe the ideas of structured programming in detail. We will show the same programs written in both old-fashioned BASIC and Structured BASIC, and compare them. In this section we describe briefly the genesis of these ideas as we applied them to Dartmouth BASIC.

As we stated earlier, the theoretical underpinnings of structured programming were in place by 1970. But it usually takes five to ten years for good ideas in computing to filter down to the "troops in the field." By the mid-1970s, most languages contained constructs needed to eliminate (almost) the GOTO statement, in short, to "do" structured programming.

Professor Stephen J. Garland led this effort at Dartmouth. He

designed constructs that were added to BASIC the Sixth through another preprocessor, which was called SBASIC, short for Structured BASIC. Actually, the PLOT preprocessor described above was combined into the SBASIC preprocessor. Our students didn't know it, but when they used SBASIC to do plotting, they were using two preprocessors.

SBASIC added four constructs to BASIC the Sixth: general LOOP, IF-THEN-ELSE, SELECT CASE, and named GOSUBs. Let's examine each in detail.

The general loop could have any one of these four forms.

```
DO WHILE I <= N
   ...
LOOP

DO UNTIL I > N
   ...
LOOP

DO
   ...
LOOP WHILE I <= N

DO
   ...
LOOP UNTIL I > N
```

These forms remain unchanged to the present time and are included in Standard BASIC.

Here is an example of the SBASIC IF-THEN-ELSE construct.

```
IF X < 3
THEN ...
     ...
ELSE ...
     ...
CONTINUE
```

Then, as now, the ELSE part was optional. Statements could appear on the same lines as the THEN and ELSE, and also on separate lines. Notice that the closing word was CONTINUE, about which more later.

An example of the SELECT CASE construct in SBASIC follows.

```
SELECT CASE n
CASE 1
   ...
CASE 2, 3
   ...
CASE 4 TO 12
```

```
    . . .
DEFAULT
    . . .
CONTINUE
```

The DEFAULT part was to be executed when none of the previous cases was satisfied. This construct used the same closing keyword, CONTINUE, as the IF-THEN-ELSE construct; we later decided this was an error and that the closing keywords for the two constructs should be different.

Finally, SBASIC allowed named internal subroutines. Instead of

```
500 GOSUB 800              'Subroutine ZILCH
    . . .
    . . .
800 REM Subroutine ZILCH
    . . .
900 RETURN
```

one could now write

```
500 PERFORM ZILCH
    . . .
    . . .
800 DEFINE ZILCH
    . . .
900 DEFEND
```

This went halfway. It allowed you to call subroutines by name rather than by line number, a big step forward, but did not allow the subroutines to have arguments. We quickly decided that allowing arguments was essential. (Remember that BASIC the Sixth already allowed *external* subprograms with arguments.)

Experience with SBASIC led us to three conclusions. First, the one feature we still had not provided our users, multicharacter variable names, was sorely needed. The limited stock of variable names provided by old-fashioned BASIC simply was not enough and did not allow programmers to name their variables in an easily readable fashion.

The second conclusion we reached was that subroutines, whether internal or external, needed to be allowed to have arguments.

Our third conclusion was that the keyword that closes a construct should be unique to that construct and should, if possible, match the opening keyword. The closing keyword CONTINUE should not have been used to close both the IF-THEN-ELSE and the SELECT-CASE constructs.

This last point is important, as it serves to distinguish Standard BASIC from several other popular languages, especially Pascal and PL/I.

To recall the Pascal style, the if-then construct would be, assuming several statements to each of the then-part and the else-part,

```
if x < 3
then begin
          ...;...
     end
else begin
          ...;...
     end
```

The grouping keywords, "begin" and "end," are also used in Pascal loops and selects, not just in the if-then-else construct.

The corresponding construct in Standard BASIC is

```
if x < 3 then
     ...
     ...
else
     ...
     ...
end if
```

Here it is clear that "end if" closes an if-then-else construct.

A recent study confirmed what many of us have long felt, that languages in which the closing keyword matches the opening keyword are easier for students to understand than languages that use begin-end or do-end to group statements.[3]

STRUCTURED PROGRAMMING, CONTINUED

In 1979 yet another version of BASIC (BASIC the Seventh) was developed at Dartmouth. By then we had recognized the need for a middle exit from a loop.

Students commonly have trouble distinguishing between two forms of the loop construct.

```
     DO UNTIL I > N
        ...
     LOOP
and
     DO
        ...
     LOOP UNTIL I > N
```

How does the beginning student know when to use the first and when to use the second? Of course, we know that the only difference is that the second form guarantees that the loop insides will be executed at least once, regardless of the values of I and N.

Having a middle exit from the loop, as in

```
DO
    ...
    (exit if the condition is met)
    ...
LOOP
```

allows us to treat the two cases above as simply special cases of the use of the middle exit, as in

```
DO
    (exit if i > n)
    ...
LOOP
```

and

```
DO
    ...
    (exit if i > n)
LOOP
```

In other words, one places the loop exit where one wants!

A more concrete example is sometimes referred to as the "read-process" problem. Suppose the user enters positive numbers and the program prints their square roots, until the user enters a 0 or a negative number. If GOTO statements were allowed, we might write

```
    DO
        INPUT number
        IF number <= 0 then GOTO 200
        PRINT sqr(number)
    LOOP
200 ...
```

Since we do not allow ourselves to use GOTO statements, we discover that there are basically two ways to code this problem using only the DO-LOOP.

```
INPUT number
DO until number <= 0
    PRINT sqr(number)
    INPUT number
LOOP
```

and

```
DO
      INPUT number
      IF number > 0 then
         PRINT sqr(number)
      END IF
LOOP until number <= 0
```

The first method is called a "process-read" loop, since the "natural" order of the two parts, INPUT and PRINT, is reversed inside the loop. Furthermore, the INPUT inside the loop gets the *next* number to be processed. Finally, the INPUT operation is duplicated, appearing once before the loop is entered and once during the loop. The second method has the defect of requiring the exit condition to be checked twice, in effect, for each time through the loop.

Still another way to tackle the read-process situation is to use flags. This approach is recommended by some adherents of structured programming.

```
LET doneflag = 0
DO
      INPUT number
      IF number > 0 then
         PRINT sqr(number)
      ELSE
         LET doneflag = 1
      END IF
LOOP while doneflag = 0
```

The flaws in all of these methods are eliminated with a middle exit from the loop. Professor Garland proposed a CONTINUE statement as part of the loop construct in BASIC the Seventh. With this addition, we can write the read-process loop more simply as

```
DO
      INPUT number
      CONTINUE while number > 0
      PRINT sqr(number)
LOOP
```

Checking interactive input for correctness requires a similar structure and is solved the same way.

```
DO
      INPUT x
      CONTINUE until (x is correct)
      PRINT "Error: enter a . . . ; retry"
LOOP
```

Another recent study showed that beginners understand programs

better when the programs are written in a language that has a middle exit from a loop.[4]

In Standard BASIC, the middle exit is given by an EXIT DO statement instead of a CONTINUE statement.

```
DO
    ...
IF <condition> THEN EXIT DO
    ...
LOOP
```

Standard BASIC also has a middle exit from a FOR-NEXT loop. Suppose we want to search a list of names for a given name.

```
FOR i = 1 to numnames
    if name$ = namelist$(i) then EXIT FOR
NEXT i
```

We think that the above construction is easier to read and understand than any other that does not use a middle exit of some form.

AMERICAN NATIONAL STANDARD BASIC

The American National Standard Committee X3J2 was formed in 1974 to develop, first, a standard for Minimal BASIC and, second, a standard for "full" BASIC. One of the first problems the committee had to grapple with was, How big should full BASIC be? Some felt BASIC should fit onto small personal computers and should therefore be quite sparse. A few even thought it shouldn't contain graphics, since several computers in those days could not do graphics. Others felt BASIC should be big enough to be useful on big machines, such as time-sharing systems. They wanted to include all sorts of extra features like fixed decimal arithmetic for business applications.

While the committee carefully deliberated and considered, the world changed. Small personal computers became big. Graphics became commonplace. Memory size and processor power grew exponentially. Backup storage (disks, both soft and hard) became faster and more capacious. The standard grew also, but each growth spurt had the effect of further delaying its official appearance. The proposed standard is quite large, but the BASIC language it defines will almost certainly fit on any major personal computer by the time the standard is finally and

formally approved—perhaps in 1985 or 1986.

We do not propose to describe the proposed standard in detail. For one thing, it contains a mass of information useful only to the specialists who write compilers and interpreters. But the examples of Standard BASIC programs in this book are written in the version of BASIC defined by the draft proposed American National Standard, with just a few additional features.[5] Figure 3.1 outlines the proposed Standard.

Figure 3.1 Outline of the Draft Proposed American National Standard for BASIC.

1. Multicharacter variable names
2. Numeric types
   a. Floating decimal preferred
   b. Lots of built-in functions
3. String types
   a. Varying length strings
   b. Lots of string functions
4. Arrays
   a. Numeric and string
   b. One-, two-, or three-dimensional
   c. MAT statements
5. Boolean expressions
6. Structured constructs
   a. Single-line IF-THEN-ELSE
   b. Multiline IF-THEN-ELSEIF-ELSE construct
   c. General DO-LOOP
   d. FOR-NEXT LOOP
   e. SELECT CASE
7. Old-fashioned GOTO and IF-THEN statements
8. Modularization tools
   a. Old-fashioned GOSUB-RETURN
   b. Single-line defined functions, internal
   c. Multiline defined functions, internal and external
   d. Subprograms, internal and external
   e. Chaining

9. Input and output
   a. Read and data
   b. Input, line input, input options
   c. Print
   d. Formatted print with USING
   e. Array read, input, and print
10. Files
   a. Display (character) files
   b. Internal, stream, and sequential
   c. Native files (optional)
   d. Array input and output to files
11. Exception handling
   a. Structured form
   b. Detached handler
12. Debugging
   a. Break points
   b. Tracing
13. Graphics (optional)
   a. Uses normalized device coordinates
   b. Input: several forms, including cursor or mouse
   c. Output: points, lines, areas, text
   d. Numerous point and line styles
   e. Color control, several forms
   f. Picture routines, transformations
14. Real time (optional)
15. Fixed-decimal (optional)
16. Editing (optional)

Graphics is optional, but certainly almost everyone will have it. Certain file-type combinations are optional, but are too complex to explain here. Real time is for process control. (Believe it or not, BASIC is an important language in process-control computers!) Fixed-decimal is for business applications. The optional editing describes simple line-numbered editing only.

In addition, the standard identifies a large number of errors that can occur during the running of a program (it calls them "exceptions"). And it recommends that error messages be printed in English!

TRUE BASIC

The draft proposed American National Standard for BASIC is just that—a standard for the language. Thus it describes the rules that must be followed in writing a program if that program is to be standard-conforming. But a standard-conforming program is useless unless there is a standard-conforming compiler or interpreter that can run the program. A standard-conforming compiler or interpreter is defined circularly as one that can correctly handle all standard-conforming programs!

In 1983, just as the technical development of the dpANS for BASIC was nearing completion, the authors, together with three colleagues, came to a shocking realization. All along we had been using good versions of BASIC, ones that kept up with the times and that made it a joy to write programs. But in 1983, we had occasions to use personal computers and came to know how terrible were the crude "street" versions that high school and college students all over the country had to use. We were appalled. (Chapter 5 explains why.)

We decided to take the bull by the horns and do something about it. We realized from past experience that writing papers or delivering talks would have no effect. The only thing that would have an effect would be to build a BASIC compiler that would run on just about every brand of personal computer. This strategy worked for the language Pascal, which is so popular in part because the folks at the University of California at San Diego built a "portable" implementation of Pascal. (This Pascal system is now called the p-system Pascal.)

The True BASIC System is the result. The True BASIC System has several parts, including a simple user interface—what the user sees and has to do in order to write and run a program. The heart of the True BASIC System is its compiler. This compiler is one of the first to be based on the draft proposed ANS standard. While the BASIC language it uses is extremely close to the standard, there are differences. We shall therefore refer to it as the True BASIC language.[6] But the reader should realize that most of the features of True BASIC are found in the standard, and vice versa.

It is not our purpose to describe all the minor differences between

the proposed standard and True BASIC, but a few are worth mentioning. One of the simplest but most important extensions is that True BASIC allows blank lines. (The closest that Standard BASIC comes to a blank line is one that has an exclamation point on it.)

In addition, True BASIC allows alternate forms for some constructs. For instance, the exception-handling construct in Standard BASIC has the form

```
WHEN EXCEPTION IN
        . . .
USE
        . . .
END WHEN
```

True BASIC allows the user to substitute the word ERROR for EXCEPTION on the grounds that it is easier to type and more natural.

True BASIC also contains additional statements for playing music, responding to single-key input, doing fast graphics for animation, and several others.

The program examples of Standard BASIC appearing in this book are all in True BASIC, unless otherwise noted.

True BASIC is a rich general-purpose language that is easy to learn and courteous to the user. Its uses range from elementary programming instruction to the construction of large and sophisticated application packages. And yet it is faithful to the principles we set down in 1964.

4
WHAT WENT WRONG?

In the years from 1964 to 1984, Dartmouth kept its own version of BASIC current with the latest ideas about language design, and added facilities needed for general applications programming. At the same time, the language remained relatively clean and free of haphazard additions. In the outside world, however, BASIC followed a more unfortunate course. In this chapter we discuss some ways in which things went wrong, in our opinion.

Dialects of BASIC proliferated. Many were written on tiny computers, so that severe compromises had to be made. Some of these compromises were ugly and violated our design principles for the language. But people got used to them, and many programs were written in these poor versions of BASIC. Meanwhile, computer memories rapidly grew bigger and cheaper. Larger versions of BASIC were demanded and produced, but they were inevitably based on the earlier versions. The ugly compromises became "features" of the language. And our design principles were repeatedly violated.

Creators of these versions were anxious to take full advantage of the hardware available to them, so they added hardware-dependent features. The result is that no two of these implementations are compatible. Programmers write not in BASIC but in "BASIC for the Apple" or "BASIC for the IBM PC." Publishers have to print different versions of a book for different computers. Authors of excellent software are frustrated when they try to "export" their creations. And those who prefer writing

in a clean, well-designed version of the language have no options at all.

Some observer has coined the phrase "Street BASIC" for this polyglot. An apt description, we believe. Vernacular street talk varies from location to location and year to year, and is full of vulgarisms not to be used in polite surroundings. Unfortunately, the same is true for BASIC. Versions differ from machine to machine and year to year, and they are full of ugly hardware-dependent features that are hard to understand and use.

Although studies of the exact genealogy of the various versions of BASIC (numbering in the hundreds) have not been carried out, it is our opinion that they evolved much like animal species, such as apes and monkeys. There were common origins, but each developmental line spawns its own progeny, and cousins bear less and less family resemblance as time goes on.

We are greatly concerned that a generation of students is growing up learning Street BASIC, an illiterate dialect of a lovely language. We feel that this is directly relevant to the problem that whereas schools now have hardware, educational software lags far behind. There have been devastating criticisms of BASIC in the literature. Unfortunately, as it applies to Street BASIC, we agree with them.

What went wrong? Many things, too many to recount completely in these pages. What were the reasons? We believe there were several. Ironically, one was the unbelievably rapid development of chip technology. In the early days of personal computers, language designers had to work with very small memories. We understand that the first BASIC produced by the Microsoft Company in the 1970s had to fit into a 4K memory. (K stands for kilobyte, which is 1024 bytes, so that 4K stands for 4096 bytes.) A remarkable programming achievement, but disastrous to the BASIC language. Compromises had to be made, to be sure, but among the compromises were many mistakes.

In the rest of this chapter we will pinpoint some of these mistakes. We will show, through examples, why we believe they were mistakes.

WHAT WENT WRONG WITH INTERPRETERS?

Oddly enough, we believe that one of the main culprits was the almost complete dependence on interpreters. The reasons are subtle, and they contradict common belief.

Two of the earliest non-GE implementations of BASIC outside Dartmouth were on the PDP-8 and HP-2000 machines, and were primarily for use in education. The companies (Digital Equipment and Hewlett-Packard) apparently wanted a simple language, one that could be implemented through an interpreter. For many reasons, they found what they wanted in BASIC. BASIC had made its mark in connection with several educational computing projects supported by the National Science Foundation. BASIC was (at that time) a small language, one that could easily fit into a small machine. It was free of some of the complicated features that are now considered essential—named subroutines with arguments, and graphics, to name just two. Finally, the language and its name were clearly in the public domain, not controlled by any standard or company.

Virtually all subsequent implementations of BASIC outside of Dartmouth followed the interpreter path.

To the surprise of many, BASIC at Dartmouth has always been, and still is, provided through a compiler. The compiler worked well for us because it was extraordinarily fast by the standards of the day. The language was designed to allow fast "one-pass" compiling entirely in core. (The word "core" is an interesting anachronism. It originally meant "magnetic core memory" and thus referred to the internal memory of the computer. But it has been fifteen years since magnetic cores were actually used for this purpose. The obsolete word lingers on.) In other systems, however, compiling usually meant very long waits before one could see the results; in those systems, the only sensible alternative was an interpreter. Interpreters possess two advantages: they can produce the first result from a small program almost immediately, and they are inherently simpler to write than compilers.

One of the big problems with interpreters is the way they detect errors. It was originally thought desirable to check each line typed in by the program as it was typed. The watchword was "instantaneous feedback." The user may realize that he or she has made a silly typing error and may be in the process of correcting it, but an error message intrudes. We want the computer to tell us about the errors we have overlooked, not the ones we can see and are about to repair. We argue that "reasonably quick feedback" together with ability to correct mistakes quickly is what was really needed. It has always seemed to us more helpful and comfortable to have the error messages appear only when we attempt to run the program, which is the way our compilers work.

There are even worse problems. One interpreter checks for syntactical errors as each line is typed in, which sounds reasonable. However, the checking is based on keywords, and not on the legal syntax

of the statement. (A keyword is any word that is a part of the language. Keywords in BASIC include IF, THEN, FOR, TO, etc. In contrast, a reserved word is a keyword that cannot be used as a variable name. Standard BASIC, which has over a hundred keywords, has only about a dozen reserved words.) The following example shows what we mean:

```
User types:    100 IF B = A THEN 500
Stored as:     100 IF B = AT HEN500
```

AT is a keyword, of course. Hobbyists and others who use this version of BASIC adapt without too much trouble, but the average user might be terribly mystified and confused. Of course, the "expert" knows enough to use one of the following instead:

```
100 IF A = B THEN 500

100 IF B = (A) THEN 500
```

To make matters worse, some of these systems advertised multiple-character variable names but examined only the first two characters of such names. For example,

```
100 LET start = 0
110 LET stop = 1
```

would first set the variable ST equal to 0, and then to 1. And pity the poor beginner who used the following,

```
120 LET stats = 5
```

which might not even be legal! ("Stats" contains the keyword AT.)

Interpreters that check lines as they are typed in, or as they are encountered when the program is run, can fail to detect certain illegal features. For example, if a rarely encountered FOR statement lacks a matching NEXT, this error may not be uncovered until months after the program is "completed."

Checking for errors as the lines are encountered when the program is run also prevents perfectly legal features from being used. For example, if an array is used only in a subroutine, it is natural to have the DIM statement there. But the interpreter complains if the DIM statement is encountered more than once, which will happen if the subroutine is called repeatedly.

Even when interpreters find mistakes, they may find them too late. Suppose you have a long (and almost correct) program that carries out an involved computation. At the end you want the answer printed, but you mistype a command as "PRNT." It will calculate for a long time and only then inform you of "syntax error", thus wasting all that time.

One of the supposed advantages of an interpreter is its ability to

allow the program to continue after an error halt. The programmer can make a few corrections to the program, or use "direct mode," and continue checking the program, without having to rerun it from the start. But one interpreter in our experience, while ostensibly allowing this mode of operation, sneakily destroyed all information about the FOR-NEXT loops and GOSUBs still in effect! What's the use of continuing the program under those circumstances?

It must be admitted that it is possible to build an interpreter that has none of the faults we mention. The truth is that interpreters are often constructed because one is short of either time or computer memory. The faults then follow as surely as night follows day.

THE PROBLEMS CAUSED BY LIMITED MEMORY

Many of the problems we identify resulted from the small amount of memory, both internal and on diskette, available to those early BASICs. Implementers devised ingenious ways to conserve valuable memory, and users adopted tricky strategies to do the same. When memory size increased, and it quickly did, the ingenious ways became ugly and the tricky strategies became just bad habits.

Because of limited memory, programs were often stored in compacted form. All letters were changed to uppercase. "Superfluous" spaces were removed. Any attempt to type programs in a format that would display its structure (as in the structured programs in this book) was defeated. Programmers were encouraged to squeeze several statements onto one line. Cryptic abbreviations were either allowed or imposed. Commenting programs was tacitly discouraged, since comments take up space. Since the result could not be read without a major deciphering effort, there was little incentive for writing readable programs. The system encouraged programmers to write totally sloppy code!

Furthermore, saved BASIC programs on such systems are not in text form and could not be easily transported over communications networks or created by general text editors. In addition, one could not write a BASIC program on such systems to prettyprint another BASIC program, or do other editing functions, because BASIC programs were not "text" files.

It became fashionable to save space by dropping the LET com-

mand. Instead of "LET a = 5", wasn't it simpler to write "a = 5"? But it destroyed the language's clean structure, that of having one statement per line, each starting with a command. How much typing does that save? We estimate 3 percent. And it creates several situations in which the user is misled. For example, in one implementation that features a WHILE command (like the "DO while" in Standard BASIC), the programmer may accidentally type

```
whilea = 1
```

when he really meant to type

```
while a = 1
```

accidentally omitting the space. The erroneous statement is accepted as legal and sets a nonexistent variable "whilea" equal to 1.

The list of such memory-conserving tricks is long: allowing you to type NEXT without a variable, thus avoiding a key check; not requiring END—so that if you lost part of your program, you might never know; assigning special meanings to multiple statements occurring on the same line—so that the physical locations of statements count and not their order.

The last trick leads to the following cryptic feature on one particular pocket BASIC we know about.

```
X = 0: Y = 3: Z = 7
```

is the same as

```
X = 0: Y = 3
Z = 7
```

but

```
IF X = 0: Y = 3: Z = 7
```

is not the same as

```
IF X = 0: Y = 3
Z = 7
```

We leave it as an exercise for the reader to figure out the explanation.

WHAT WENT WRONG WITH ERROR REPORTING?

There are many horror stories about the error-checking provided by these early BASICs.

First of all, most of them print error messages like "syntax error on line 230." Not a hint as to what kind of error or where the error is on the line. We are used to a compiler that will give a specific error message and point to the exact spot in the line where the error is.

Giving reasonable error messages is not hard. A few days of extra effort is all that would be needed to produce succinct yet clear English phrases that tell exactly what is wrong. Is that asking too much?

WHAT WENT WRONG WITH LOOPS?

A bad mistake in some versions is the way a FOR loop is implemented. Suppose the loop started with

```
200 FOR X = 1 TO N
```

If N = 3 the loop is carried out three times, if N = 1 then once, but what should it do if N = 0? Dartmouth BASIC has always recognized that such loops are meant to be executed zero times (i.e., skipped). (FORTRAN worked the other way. The FORTRAN DO loop, which is like the BASIC FOR-NEXT, always carried out the instructions inside it, regardless.) This is crucial for several well-known mathematical algorithms to work correctly in the case N = 0. Furthermore, the ANSI standard for Minimal BASIC, which was approved in 1978 and known in content as early as 1976, specified zero times for the above example. Certain versions of BASIC chose instead to follow the FORTRAN model. We have seen many programs wherein such FOR loops are surrounded by an IF-THEN to treat "zero times" as a special case:

```
100 IF N = 0 THEN 500
200 FOR I = 1 TO N
300       ...
400 NEXT I
500 ...
```

Over the years we have received many complaints that programs we have published "do not work." We had to puzzle out in each case just what was wrong with the particular implementation of the language. By far the commonest problem was the one just described.

It is ironic that the ANSI committee charged with maintaining the FORTRAN standard is considering changing the DO loop to perform like Standard BASIC's FOR loop!

WHAT WENT WRONG WITH GRAPHICS?

One of the key principles that always guided us in implementing BASIC was that the user should not have to learn the peculiarities of the hardware. Unfortunately, this principle has been almost universally violated.

There are many examples of the language being spoiled by hardware dependence. We hate even to mention having to give memory addresses in hexadecimal. (Hexadecimal is a number system to base 16, which requires one to use A, B, C, D, E, and F as "digits.") Even the earliest version of FORTRAN freed us from such horrors!

We mentioned that one feature of FORTRAN we were happy to eliminate was having two kinds of numbers. Some versions of BASIC have four kinds of real numbers! They not only distinguish integers from floating point numbers, but have single-precision and double-precision numbers. The four types are distinguished by special symbols at the end of the variable names. Thus, if you request double precision, you will find that your variable names have been changed by the addition of funny symbols!

But the worst hardware dependence is in graphics. Although many versions of BASIC allowed users to draw colorful pictures on the screens of personal computers, only the crudest tools were provided. To dramatize the distinction between a good and poor version of BASIC, we will describe our experience exactly as it happened.

We will draw a simple picture. Let us start with an easy one: a triangle starting at the origin with base 2 and altitude 2. We first show how it is done in True BASIC.

We define our work area on the x-axis as from −2 to 5 and on the y-axis as from −4 to 4 (SET WINDOW). Fine, that works. But we want it in red (SET COLOR). And if we are going to draw several figures, it might be nice to leave a border around the screen (OPEN #1: SCREEN). Finally, we plot the edges that join the points (PLOT LINES). Our complete program follows:

```
OPEN #1: SCREEN .1, .9, .1, .9          ! Leave border
SET WINDOW -2, 5, -4, 4                 ! Coordinates
SET COLOR "red"                         ! Draw in red
PLOT LINES: 0,0; 2,0; 1,2; 0,0          ! Triangle
END
```

That was easy!

Now let us do the same simple task in BASICA, the version of the language IBM supplies on its PC. We can draw lines with a LINE statement. Fine. But we are told that the dots in the horizontal direction are numbered 0 to 639, and in the vertical direction 0 to 199. Let's see, if x = −2 corresponds to 0 and x = 5 to 639, y = −4 to 0 and y = 4 to 199, then what corresponds to the point (1,2)? Excuse us for a while . . . .

We are back. Nice bit of algebra. Any mathematician could do it. We begin to see our picture. But why is the triangle upside down? More reading of the manual. Aha! While the horizontal points are numbered from left to right, on the vertical axis the big numbers are at the bottom! Evidently, IBM chose to follow the television industry, which scans a TV picture starting at the top, rather than us mathematicians, who prefer to use the Cartesian plane we all learned in high school. So we recompute the y-coordinates:

```
LINE (183,99)-(369,99)
LINE -(274,50)
LINE -(183,99)
```

Now the triangle looks right. Wait, we had no END statement. Apparently it doesn't care. Look at the program. Can you see that it is an isosceles triangle? We can't.

Now we want to add color. Look up the magic number for setting color, and insert a SCREEN statement. Then look up which palette has red in it, and issue a COLOR statement. Finally, add the magic number of red (within the palette) to the end of each LINE statement. We run our program again. What happened? It is in red all right, but the triangle has moved over to the right edge of the screen, and we lost a portion of it. Why? All we did was ask for it in red.

Back to the manual. It seems that if we draw in color, the horizontal points are numbered from 0 to 319, not to 639. Why didn't we think of that! OK, recompute the x-coordinates. Now we have a nice red triangle.

Last step. Let us leave a margin on the screen: Let's see, the horizontal points should now go from 31.9 to 287.1. . . . We give up. After all, we are only mathematicians!

If you feel that you need a computer to do all these messy calculations, we agree with you. You do need a computer to do all these calculations. But you have a computer! So why does BASICA make you do all the terrible arithmetic? Because it is a very poorly designed language. Why do millions put up with this? Because they do not realize

that one can provide much friendlier graphics tools. Our students and teachers deserve far better.

WHAT WENT WRONG WITH STYLE?

Did style go out of style? Where did all those little details go, the details that help one read programs? Details like lowercase letters, indentation, blank lines, and on-line comments. Why were they excised and exiled?

Would you read this book if only uppercase letters were used, if there were no indents indicating new paragraphs, no special treatment of displays of sample programs, or if there was inconsistent spacing between words or sentences? Of course not. But that is exactly what most versions of Street BASIC impose.

The following example shows what you have to do in one popular BASIC to use on-line comments or blank lines.

```
100 LET X = 3 : REM INITIALIZE X
200 :
300 FOR I = 1 TO 5
```

This version mandates all spacing not in REM statements, so that when you type

```
100 FOR i = 1 to 10
110     LET x = x + a(i)
120 NEXT i
```

you get

```
100 FOR I = 1 TO 10
110 LET X = X + A ( I )
120 NEXT I
```

We have known all along that "styling" the listing of a program is just about as important as anything else in helping to read and understand it. We insist on being able to use blank lines, being able to use indentation the way we want it, and being able to use either uppercase or lowercase letters, or both, when we want. Our first recorded (in print) use of indentation to enhance the appearance of a program occurred in 1965 (see chapter 2). Our intuition was confirmed scientifically in 1983 (see note 6 in chapter 2.)

THE PROBLEM WITH STANDARDIZATION

Standardization is one important way to prevent the chaotic evolution we have seen with BASIC. But standardization failed for so long for two reasons. The first is that standardization did not start until 1974, ten years after the invention of BASIC. In this we are partly to blame. If we had realized earlier that BASIC would become so widespread, we might have worked harder to get the ball rolling sooner. But we didn't. And even in 1974 we still had no idea that personal computers would take over the way they have.

The second reason is that, once the standards process had begun, the rules changed. The committee first had to develop a standard for Minimal BASIC, and spent two years on the task. By the time it appeared in print (1978), it was obsolete. The committee had barely begun with the standard for full BASIC when structured programming and interactive graphics became essential. Finally, in the early 1980s, it became obvious that computer memory was very, very cheap and that language limitations dictated by small memories would disappear long before any standard appeared.

WHAT ARE WE DOING ABOUT IT?

People have asked us why we did not choose to control the development and use of BASIC. There were several reasons. We made a decision to put the language into the public domain so that it would be widely used. This precluded our imposing controls on the use of the language. Second, we did not foresee that the language we spawned would be accepted so widely around the world, although in forms we might not approve of. Third, we felt that any attempt at controlling the language would lead to less acceptance. Indeed, there was some evidence at the time to support this contention. If we had denied, by some mechanism not known to us, the right of others to copy and modify the language, the most likely course would have been for others to develop variations but not call them BASIC. The connection to the original BASIC at Dartmouth would then have been lost forever.

But what caused the problem—the rapid growth of personal

computers—will bring about the solution. Computers within the price reach of almost anyone now have enough memory and processing power to support truly high-quality implementations of BASIC. No longer will it be necessary for kids to learn lousy versions of the language. We have dedicated ourselves to the task of providing schools with a BASIC we can all be proud of.

5
STRUCTURED PROGRAMMING

The best approach to writing a large program is to divide the task up into several small tasks. Each individual task should serve a clearly defined purpose, and it should be possible to debug the small tasks one at a time. It is also important that the logical relation among tasks be clearly shown. The result is a program that is easy to read, easy to debug, and easy to modify.

This method of writing programs is applicable to any programming language. However, some languages allow such "structuring" of programs in a simpler and more natural way. Structured programming was possible in earlier versions of BASIC, as was shown in our book *BASIC Programming*, third edition. It is the purpose of this chapter to show that Standard BASIC is far superior for the construction of well-structured programs. We will use a series of examples to illustrate the advantages of a fully structured language. It is not our intention to teach programming; we will assume that the reader is familiar with programming in some version of BASIC.

In each case we will first write a program in an older version of BASIC and then show it rewritten in True BASIC. We hope to convince the reader that the True BASIC program is

1. Much easier to read.
2. Easier to debug.
3. Easier to modify.
4. No harder to write. Indeed, once the new structures are fully understood, the programs are easier to write in True BASIC.

SUM

```
100 REM Sum of consecutive integers
200 LET sum = 0
210 FOR n = 1 to 75
220 LET sum = sum + n
230 NEXT n
240 PRINT sum
300 END
```

Our first example adds the integers from 1 to 75. It is a very simple program. And since the FOR-NEXT structure was well designed even in the earliest version of BASIC, no significant improvement is possible.

We can make the program slightly more readable by the use of blank lines, and by indenting the body of the FOR-NEXT structure.

SUM-S

```
100 REM Sum of consecutive integers
110
200 LET sum = 0
210 FOR n = 1 to 75
220     LET sum = sum + n
230 NEXT n
240 PRINT sum
250
300 END
```

Using blank lines in programs is similar in purpose to using them in books to separate paragraphs. The extra space makes the text more readable.

The advantage of indenting the body of a structure will be much more obvious in more complex examples.

Although FOR-NEXT loops were available from the beginning, another important type of loop was missing from earlier versions of the language. We illustrate its use through two simple examples.

LOOP

```
100 REM Calculates products
200 INPUT a, b
210 PRINT a*b
220 GOTO 200
300 END
```

In this simple program we ask the user to supply two numbers, we multiply them together, and print the answer. Then we "loop" back to ask for two more numbers.

The program is so simple, it is hard to argue that there is anything wrong with it. But it does not show clearly that we have written a loop. You have to read to line 220 to discover that line 200 is the beginning of a loop. The weakness of this approach would be much clearer in a long program. Consider a five-page program that contains near its end a GOTO 200. You would then be forced to reread the whole program from the perspective that it is a very long loop!

A FOR-NEXT loop is not suitable for this program. The appropriate looping structure is DO-LOOP.

LOOP-S

```
100 REM Calculates products
110
190 DO
200     INPUT a, b
210     PRINT a*b
220 LOOP
230
300 END
```

So far all that we can claim is that we can clearly see that a loop is involved. The advantages will be evident in more interesting examples.

One shortcoming of either program is that it never stops. We might attempt to improve it by asking the user after each example whether he or she wishes to continue.

LOOP2

```
100 REM Calculates products
200 INPUT a, b
210 PRINT a*b
220 PRINT "More";
230 INPUT answer$
240 IF answer$ = "yes" then 200
300 END
```

Again the program is short and simple, but it does not make it clear that line 200 starts a loop. The True BASIC version is just as simple, and it makes the structure clear.

LOOP2-S

```
100 REM Calculates products
110
190 DO
200     INPUT a, b
210     PRINT a*b
220     PRINT "More";
230     INPUT answer$
240 LOOP while answer$ = "yes"
250
300 END
```

This version shows that we may loop "while" a certain condition holds. Alternatively, we could have used

```
240 LOOP until answer$ = "no"
```

We will see more interesting uses of the DO-LOOP in later examples.

Next we turn to "branching." Any computer language must allow a choice of procedures that depend on the outcome of previous calculations. If a certain event happens, we proceed in one way; otherwise we must do something else. The IF statement allowed branching even in the first version of BASIC. But it only allowed a primitive and somewhat awkward choice. We will use a well-known example where we must choose among three alternatives.

QUAD

```
100 REM Quadratic equation
200 PRINT "Coefficients";
210 INPUT a,b,c
220 LET d = b*b - 4*a*c
300 IF d <= 0 then 360
310 LET s = sqr(d)
320 PRINT "The roots are:"
330 PRINT (-b+s)/(2*a)
340 PRINT (-b-s)/(2*a)
350 GOTO 500
360 IF d < 0 then 400
370 PRINT "The root is:"
380 PRINT -b/(2*a)
390 GOTO 500
400 PRINT "No real roots."
500 END
```

Our problem is to find the real roots of a quadratic equation

$$ax^2 + bx + c = 0$$

This is a problem we learned to solve in high school, by means of the "quadratic formula." The nature of the solution depends on the "discriminant," the quantity called d on line 220. If d is greater than 0, then there are two different real numbers x which are solutions. If d = 0, then there is one solution. If d is negative, the roots are complex numbers. We limit ourselves to saying that there are no real roots.

If one glances at the program, no structure is evident. It takes careful reading to realize that we are treating three different cases. Nor is the nature of the cases obvious. The first case is d > 0. But the IF statement on line 300 tests for d <= 0. In a sense, we are forced to apply the test backward. We do not test on what we want to do next; instead we skip on to the other cases. We also have the nuisance of having to insert the GOTO 500 lines, so that when we finish one case, we skip over the remaining ones. This is the first example in which the advantages of the True BASIC version will be evident.

QUAD-S

```
100 REM Quadratic equation
110
200 PRINT "Coefficients";
210 INPUT a,b,c
220 LET d = b*b - 4*a*c
230
300 IF d > 0 then
310    LET s = sqr(d)
320    PRINT "The roots are:"
330    PRINT (-b+s)/(2*a)
340    PRINT (-b-s)/(2*a)
360 ELSEIF d = 0 then
370    PRINT "The root is:"
380    PRINT -b/(2*a)
390 ELSE
400    PRINT "No real roots."
410 END IF
420
500 END
```

The three cases are identified by IF, ELSEIF, and ELSE. The end of the branching structure is marked by END IF. The tests are carried out in the natural order (not backward). And there is no need for the GOTO statements, since only one of the three cases is carried out for any given value of d.

We hope that the reader will agree that this version is much easier to read. We also suggest that it is easier to write.

A very important feature of structured programming is the ease with which improvements can be added. To illustrate this we will add to this pair of programs the capability of solving several quadratic equations. We will enclose the main part of the program in a loop, and ask the user each time whether he or she wishes to continue.

QUAD2

```
100 REM Quadratic equation
200 PRINT "Coefficients";
210 INPUT a,b,c
220 LET d = b*b - 4*a*c
300 IF d <= 0 then 360
310 LET s = sqr(d)
320 PRINT "The roots are:"
330 PRINT (-b+s)/(2*a)
340 PRINT (-b-s)/(2*a)
350 GOTO 450
360 IF d < 0 then 400
370 PRINT "The root is:"
380 PRINT -b/(2*a)
390 GOTO 450
400 PRINT "No real roots."
450 PRINT
460 PRINT "More";
470 INPUT answer$
480 IF answer$ = "yes" then 200
500 END
```

The addition of lines 450–470 is simple. But again, the program suffers from the need to reach line 480 before we realize that we are looping. Here we also had the additional problem that, upon adding lines 450–470, GOT0 500 had to be changed to GOTO 450 in two places. It would be very easy to miss one of these.

There is a further danger. If we decided to add a line 190 at the beginning of the loop, we would have to remember to change line 480. In each of these cases the weakness is that line numbers as addresses for GOTO or IF statements point to a physical location, not to a logical address. Thus, in QUAD, 500 happened to be the first instruction after the end of the IF structure. But once 450–470 are added, that is no longer true. And 200 happens to be the beginning of the loop, but the addition of a line 190 could change that. These problems do not arise in the structured version of the program.

QUAD2-S

```
100 REM Quadratic equation
110
190 DO
200     PRINT "Coefficients";
210     INPUT a,b,c
220     LET d = b*b - 4*a*c
230
300     IF d > 0 then
310        LET s = sqr(d)
320        PRINT "The roots are:"
330        PRINT (-b+s)/(2*a)
340        PRINT (-b-s)/(2*a)
360     ELSEIF d = 0 then
370        PRINT "The root is:"
380        PRINT -b/(2*a)
390     ELSE
400        PRINT "No real roots."
410     END IF
420
450     PRINT
460     PRINT "More";
470     INPUT answer$
480 LOOP while answer$ = "yes"
490
500 END
```

After adding 450–470, we only needed to add the obvious DO-LOOP statements, and no further modifications were necessary. We did not have to change line number addresses. And if we wanted to add a new first line to the loop, we would clearly put it after DO, and no other change would be needed.

This example also shows how much indentation can help to display the structure of the program. It is obvious that the IF structure is contained within the DO-LOOP. And the three cases of IF stand out.

We managed to write the program without using line numbers as addresses, preventing certain common errors that occur when programs are renumbered. In True BASIC we need never use line numbers as addresses. Indeed, line numbers are used only as a convenience for the editing of programs. Programs without line numbers are legal. We shall have more to say on this topic later.

DICE

```
100 REM Simulate 10 plays of a dice game.
200 FOR game = 1 to 10
210 LET die1 = int(6*rnd + 1)
220 LET die2 = int(6*rnd + 1)
230 LET dice = die1 + die2
240 PRINT dice;
300 ON dice-1 goto 400,400,600,600,600,500,600,600,600,
    500,400
400 PRINT "You lose"
410 GOTO 700
500 PRINT "You win"
510 GOTO 700
600 LET point = dice
610 LET die1 = int(6*rnd + 1)
620 LET die2 = int(6*rnd + 1)
630 LET dice = die1 + die2
640 PRINT dice;
650 IF dice<>7 and dice<>point then 610
660 IF dice=7 then 690
670 PRINT "You win"
680 GOTO 700
690 PRINT "You lose"
700 NEXT game
900 END
```

This program illustrates a choice among several different cases. We use the well-known dice game called shooting craps.The player rolls a pair of dice and adds up the points. (This is carried out on lines 210–240 using the random number generator "rnd." Two dice are rolled, the sum is "dice," and this sum is printed so that we may follow the course of the game.) If the sum is 2, 3, or 12, the player loses. If it is 7 or 11, the player wins. For any other outcome the sum becomes the player's "point." The goal then is to repeat the point before a sum of 7 is rolled. The three cases correspond to 400, 500, and 600. At 600 dice are rolled again (610–640) and rolling continues until the sum is 7 or equal to "point."

There are eleven possible outcomes for the sum of two dice, namely 2 through 12. The ON statement allows us to make an eleven-way choice. Unfortunately, it is the most awkward structure in BASIC. It suffers from all the difficulties of using line numbers as addresses. In addition, line 300 is very hard to decipher! The result is a program whose structure is obscure.

DICE-S

```
100 REM Simulate 10 plays of a dice game.
110
200 FOR game = 1 to 10
210      LET die1 = int(6*rnd + 1)
220      LET die2 = int(6*rnd + 1)
230      LET dice = die1 + die2
240      PRINT dice;
300      SELECT CASE dice
350      CASE 2,3,12
400           PRINT "You lose"
450      CASE 7,11
500           PRINT "You win"
550      CASE else
600           LET point = dice
605           DO
610              LET die1 = int(6*rnd + 1)
620              LET die2 = int(6*rnd + 1)
630              LET dice = die1 + die2
640              PRINT dice;
650           LOOP until dice=7 or dice=point
660           IF dice=point then PRINT "You win" else
                                      PRINT "You lose"
670      END SELECT
700 NEXT game
710
900 END
```

The ON statement is replaced by the elegant SELECT structure. It allows us to divide the program into clearly labeled cases, based on the value of the variable "dice" (line 300). The nature of the three cases is clearly displayed. The end of the structure is also clearly indicated, by END SELECT.

We are also able to make the last case (trying to "make a point," i.e., repeat the first value) much clearer, by using the DO-LOOP. And line 660 is much simpler than lines 660-690 in DICE.

This is our best example to demonstrate that a well-designed structure (SELECT) can make programs more readable and also makes programming easier.

In each of these two programs there is a repetition of code (lines 210–240 and 610–640). This repetition can be eliminated by means of a subroutine, as will be seen in the next pair of examples.

DICE2

```
100 REM Simulate 10 plays of a dice game.
200 FOR game = 1 to 10
210 GOSUB 800
300 ON dice-1 goto 400,400,600,600,600,500,600,600,600,
    500,400
400 PRINT "You lose"
410 GOTO 700
500 PRINT "You win"
510 GOTO 700
600 LET point = dice
610 GOSUB 800
650 IF dice<>7 and dice<>point then 610
660 IF dice=7 then 690
670 PRINT "You win"
680 GOTO 700
690 PRINT "You lose"
700 NEXT game
790 GOTO 900
800 REM Roll dice
810 LET die1 = int(6*rnd + 1)
820 LET die2 = int(6*rnd + 1)
830 LET dice = die1 + die2
840 PRINT dice;
850 RETURN
900 END
```

The code for rolling a pair of dice is placed in lines 800–850. To invoke this subroutine we write GOSUB 800. The RETURN instruction then returns us to the statement following the GOSUB. Thus DICE2 performs exactly as DICE. But the program is shorter and better organized. In a more complicated game, rolling dice might occur at several different places, allowing even greater savings.

While this is a better way of organizing the program, GOSUB has a number of shortcomings. First of all, GOSUB 800 gives us no clue as to what the subroutine being invoked does. In a long program having many subroutines, the programmer needs a table of contents for the subroutines. There are the usual problems with line number addresses as well. More important, the programmer could jump into the middle of a subroutine (e.g., GOSUB 830). Although there are occasions when this seems convenient, this is often the source of errors.

Finally, line 790 is essential to avoid "falling into a subroutine." It is as awkward and as easy to forget as the GOTO statements necessary with the old version of IF.

DICE2-S

```
100 REM Simulate 10 plays of a dice game.
110
200 FOR game = 1 to 10
210       CALL roll
300       SELECT CASE dice
350       CASE 2,3,12
400             PRINT "You lose"
450       CASE 7,11
500             PRINT "You win"
550       CASE else
600             LET point = dice
605             DO
610                CALL roll
650             LOOP until dice=7 or dice=point
660             IF dice=point then PRINT "You win" else
                                              PRINT "You lose"
670       END SELECT
700 NEXT game
710
800 SUB roll
810       LET die1 = int(6*rnd + 1)
820       LET die2 = int(6*rnd + 1)
830       LET dice = die1 + die2
840       PRINT dice;
850 END SUB
860
900 END
```

In True BASIC, subroutines have names and are invoked by a CALL statement. The statement "CALL roll" gives much more information about the routine than "GOSUB 800." The subroutine is clearly identified, starting with SUB and ending with END SUB. It is a self-contained piece of code, which may be placed anywhere in the program. The user cannot accidentally "fall into"a subroutine.

Most readers will agree that this is a big improvement over GOSUB. And it is certainly easy to use. But the identification of subroutines as self-contained pieces of code is much more significant. It leads to powerful new options, which lie at the heart of structured programming. We shall discuss these further enhancements of subroutines later.

Let us pause to consider what we may learn from these examples. It is often claimed that the older versions of BASIC are "unstructured." That is not strictly true. The statements FOR-NEXT, IF, ON, and GOSUB allow a reasonable degree of structuring of programs. But what was accepted as reasonable fifteen years ago no longer meets the standards set by modern computer science.

We have seen that the more sophisticated version of IF, replacement of ON by SELECT, the addition of the DO-LOOP, and named subroutines allow us to display the structure of programs much more clearly. And those of us who made the transition find it easier to program in the fully structured version of the language. We now have several years' experience in teaching students structured BASIC, and we find that they make significantly fewer mistakes and can advance to large programs much more rapidly. By the standards we routinely impose on our students today, the programs of a decade ago are unacceptable!

When we first designed BASIC, we were fortunate enough to make many decisions that stood the test of time. That accounts for the enormous popularity of the language. But we now believe that we made one very serious mistake. The mistake concerns one use of line numbers.

When we decided that each line should start with a number, we had three different uses in mind:

1. Ease of editing
2. Simple error messages that furnished the number of the line containing the error
3. An address for "jumps" in GOTO, IF, ON, and GOSUB statements

The first two have stood the test of time; the third has not.

Line numbers allow easy correction (just retype the line with the same number), insertion (type line with new number), and deletion (line number alone). Until the coming of personal computers, this capability made programming vastly easier. Now that personal computers provide "screen editing," users may prefer to omit line numbers.

However, we are not sure that all users will. Some changes may still be simpler by line number, and having line numbers in error messages is very convenient. Therefore, we have kept the option, and True BASIC allows programs with or without line numbers.

Originally, we thought that it would be natural to use line numbers as addresses for jumps. Unlike the rest of the language, which was based on logical structure, this feature depended on the physical location (the line number) of a statement. Line numbers are frequently changed, and programmers renumber their programs. Unless renumbering is carried out by means of a sophisticated line editor, the chance of error is very great. And deletion of one line may result in an illegal program, for example, when the program contains a GOTO 800 and line 800 is deleted. The addition of a line may result in a program that is legal but incorrect if we fail to change the line number address in a GOTO, IF, ON, or GOSUB statement.

There is a certain amount of time needed to make a transition in programming style, but the effort is not great to learn to use the new, powerful versions of IF, to replace ON by SELECT, and to use CALL in place of GOSUB. And anyone who has tried both methods will prefer the new, fully structured version. That leaves one statement having no precise analog in structured languages—namely, GOTO.

Substantial literature exists to argue the dangers of GOTO. Allowing the programmer to jump from any place within the program to any other destroys the spirit of structured programming. It is also the source of several kinds of errors.

We ourselves, while strongly supporting the development of a fully structured BASIC, were initially skeptical about the need to eliminate all GOTO statements. We produced examples where, although GOTO could be avoided, the resulting code was unnatural. Such arguments led to the inclusion of additional features in Standard BASIC, which remove all arguments in favor of GOTO. One example is the "middle exit" from a loop, which we have already discussed. Today, we do not allow our students to use line numbers as addresses, which prohibits the use of GOTO and the other statements discussed above.

However, Standard BASIC has retained these statements. Hence they are included in True BASIC. This should ease the problems of transition. While the reader is learning to write structured programs, he or she should feel free to use the old versions where it is not clear how to use the new ones. We predict that the old IF, ON, and GOSUB will disappear rapidly, but a few GOTOs will remain. With experience the temptation to use GOTO will also diminish. Eventually programmers will eliminate line numbers completely.

Since our remaining programs are fully structured, we will show them without line numbers.

While subroutines are useful for removing duplication of code, modern programming theory assigns an even greater role to them. A good

program should be written in small, easy-to-read, and easy-to-debug "segments." And the most important tool for segmenting a program is the subroutine.

We use the subroutine whenever a program is too long, to divide it into smaller and more manageable pieces. This is the case even if the subroutine is called only once. And if the subroutine is quite long, it in turn is segmented, usually by using other subroutines.

We teach our students to write a "skeleton" first and then write the subroutines that put flesh on the skeleton. The technical name for this approach is "top-down programming."

As an illustration, consider the problem of counting the number of words in this chapter. We will assume that the chapter is stored in a file on our computer (it is!). The program must first find out the name of the file and gain access to it ("open" the file). Then it should read the chapter one line at a time, counting the words in each line, and adding these to a specified variable, "count" in our example. The skeleton may look as follows:

```
CALL getfile
LET count = 0
DO while more #1
   LINE INPUT #1: line$
   CALL words
LOOP
PRINT "Number of words:"; count
```

We will show the full program in the last chapter. There we will see that the subroutine "getfile" is quite simple. But the subroutine "words" is further segmented by calling another subroutine.

Another important tool for segmenting is the defined function. A number of functions, such as sqr, sin, log, are built into BASIC. But even early versions of the language allowed the user to define additional functions. For example,

```
DEF f(x) = x ^ 3 - 3*x ^ 2 + 2
```

defines a simple polynomial. After the definition, any time f(x) is encountered in the program, the defining formula is evaluated. Thus f(0) = 2 and f(1) = 0. Suppose that we wished to define the average of two numbers. There are two commonly used averages in mathematics. One is the ordinary average, or mean. The other is the geometric mean, for which we multiply the numbers and take a root of the product.

```
DEF mean(a,b) = (a+b)/2
DEF geo_mean(a,b) = sqr(a*b)
```

Once we have introduced these two definitions, the computer can

find that mean(2,8) = 5 and geo_mean(2,8) = 4. It is a famous theorem that the latter is smaller than the former (they are equal only if a = b).

These are examples of one-line definitions. But many functions require more complex definitions. Suppose that we have n numbers in a list and wish to compute their mean. The following multiline definition will accomplish this:

```
DEF mean(list(),n)
    LET sum = 0
    FOR i = 1 to n
        LET sum = sum + list(i)
    NEXT i
    LET mean = sum/n
END DEF
```

The structure starts with DEF and ends with END DEF. The definition is in between. We add up the n numbers in "list," and the answer "mean" is set equal to the sum divided by n. The arguments of the function are on the DEF line. The first "list" is followed by parentheses to indicate that it is a list or array rather than a single number. The arguments are called "parameters."

The geometric mean of a list may be computed similarly:

```
DEF geo_mean(list(),n)
    LET product = 1
    FOR i = 1 to n
        LET product = product * list(i)
    NEXT i
    LET geo_mean = product ^ (1/n)
END DEF
```

These examples show how important parameters are. The definition of "mean" is for any two numbers *a* and *b*, or for any list of numbers. Once we have made subroutines into freestanding segments of code, we can enrich them by allowing parameters. For example, to interchange two numbers, three instructions are necessary: one must remember the first, then replace it by the second, and then replace the second by the remembered original value of the first. We may find it useful to have the short subroutine

```
SUB swap(x,y)
    LET temp = x
    LET x = y
    LET y = temp
END SUB
```

We can then intechange numbers by calls such as

```
CALL swap(a,b)

CALL swap(first,second)
```

The next major idea in structured programming is that often-used routines, such as "swap" and "mean," need not be part of your program. They can be written once and used by many programs—and many users. A collection of such routines is known as a library of programs. The main program will contain a LIBRARY statement, stating which library (or libraries) it uses, and all the routines in the particular library become available to it.

We are going to illustrate the building of a small library. Our examples so far have dealt only with numbers, but BASIC can deal with verbal information as well. We shall discuss this use of BASIC and build a small library of routines, which we will call "STRINGS" (see Figure 5.1).

BASIC deals with two kinds of objects: numbers and strings. A string may consist of a single letter, a word, a sentence, a line of a program, or even a meaningless collection of symbols. One designates a string by using a variable ending with a dollar sign, such as "word$" or "name$."

One cannot apply arithmetic operations to strings, or take the logarithm of a string. But BASIC does contain special string operations and functions. For example, we may wish to extract a portion of a string. If name$ = "True BASIC," then name$[3:8] = "ue BAS" (characters 3 through 8). The functions "lcase$" and "ucase$" change a string into lowercase and uppercase letters. This is very convenient when we wish to compare two strings. Suppose that the user is asked a question and replies in the affirmative. The answer might be "YES" or "yes" or "Yes." We can test the answer by

```
IF lcase$(answer$) = "yes" then
```

and we will catch all three versions of the answer. The function "abbr$" in our library abbreviates to three letters and takes lowercase. This is frequently used when users type commands to a program, allowing full commands or three-letter abbreviations, and allowing lowercase, uppercase, or a mixture.

The function "len" supplies the length of a string, that is, the number of characters. All characters count, including punctuation marks and spaces. The most useful string function is "pos," which finds the position of one string within another. More precisely,

```
pos(a$, b$, ch)
```

starts looking for string b$ within a$, starting at character number "ch".

If it is present, the value of the function is the number of the first character of b$. That is, the function points to the beginning of b$. If b$ does not occur after character ch, the answer is 0.

Some examples will make this clearer. For each we show what a$ and b$ are, what ch equals, and the value of pos:

a$	b$	ch	pos
scatter	cat	1	2
matter	cat	1	0
Dartmouth	t	1	4
Dartmouth	t	6	8
Dartmouth	t	9	0

Let us return to our small library "STRINGS." The second definition in it is of the function "vowel" that counts the number of vowels in a string. We will let the reader decipher the routine. To help, we have added comments. Comments may be added at the end of a line in BASIC by using the exclamation point to indicate that the rest of the line is a comment.

The subroutine "rep" is a very useful one. It replaces all occurrences within a$ of the string b$ by the string c$. For example, if a$ is a line of a program, b$ = "x" and c$ = "y", then all letters x in the line are replaced by y. The code is short but tricky. The variable ch points to a character in a$. We start at the beginning (ch = 1) and loop until our task is completed. We see whether b$ occurs after character ch. If not (ch becomes 0), we EXIT DO, and are finished. Otherwise ch2 will point to the end of b$ and we replace b$ by c$. Then we set ch to point to the character after the c$ we have just inserted, and try again. The reader is urged to work the following example, using a piece of paper to keep track of the values of variables:

```
a$ = "The dog is the nicest dog."
b$ = "dog"      c$ = "horse"
```

The library begins with the statement EXTERNAL to indicate that these procedures are outside a main program. To use the library, you insert two statements in the main program:

```
LIBRARY "STRINGS"
DECLARE DEF abbr$, vowel
```

BASIC needs the latter to know that the names stand for functions. Since subroutines are always used by means of a CALL statement, no declaration is necessary for them.

FIGURE 5.1 Library of String Routines.

```
STRINGS
EXTERNAL                                ! Library of string routines
DEF abbr$(a$) = lcase$(a$[1:3])         ! Lower case, 3- letters
DEF vowel(a$)                           ! Count vowels in string a$
    LET sum = 0                         ! Number of vowels
    FOR ch = 1 to len(a$)               ! Look at each character
        SELECT CASE lcase$(a$[ch:ch])   ! One char., l.c.
        CASE "a", "e", "i", "o", "u"
           LET sum = sum + 1            ! Count vowel
        CASE else
           ! Do nothing
        END SELECT
    NEXT ch
    LET vowel = sum                     ! Value of function
END DEF
SUB rep(a$,b$,c$)                       ! Replace all b$ in a$ by c$
    LET ch = 1                          ! Character position
    DO
       LET ch = pos(a$,b$,ch)           ! Is b$ present?
       IF ch=0 then EXIT DO             ! No, get out
       LET ch2 = ch - 1 + len(b$)       ! Point to end of b$
       LET a$[ch:ch2] = c$              ! Replace by c$
       LET ch = ch + len(c$)            ! Point past c$
    LOOP                                ! See if more needs to be
                                                          done
END SUB
```

We hope that these examples will give the reader a feeling for the power and convenience of structured programming. A collection of structured examples written in True BASIC will be found in the last chapter.

6
TRUE BASIC AND OTHER LANGUAGES

How does True BASIC compare with other languages? Is it the case, as some avow, that learning BASIC teaches bad habits and that students should learn some other programming language like LOGO or Pascal? Since many programming jobs require FORTRAN or COBOL, shouldn't they be taught instead? Does Ada, the new Defense Department language, change the picture? Or is True BASIC a respectable member of the family of important computer languages? In this chapter we will consider the facts.

True BASIC is the version of BASIC we use in this chapter to compare with the other languages. But practically all that we say about True BASIC applies as well to Standard BASIC.

THE BAD-HABIT ACCUSATION

The defense of BASIC here is simple. The indictment applies to Street BASIC, which depends on GOTOs and IF-THENs to line numbers. Undisciplined use of GOTOs and IF-THENs leads to what some people call "spaghetti code." Take a listing of a Street Basic program and draw a line from each GOTO statement to the target line; the result often resembles a bowl of spaghetti.

It is possible, and not too hard, to write well-structured programs in Street BASIC. We will not show the reader how to do this because that skill is not needed now that we have True BASIC.[1]

It is also possible, and not too hard, to write badly structured programs in a structured language such as True BASIC or Pascal. There are actually two factors at work here. To write a good program, one needs to start with a good design. This can be as simple as just thinking for a few moments before starting to code (with a small program), or it can mean spending weeks planning which subroutines are needed (with a huge program.) This is an extremely important topic, but one that we do not cover in this book.

The other factor concerns the features of the programming language to be used. The language should make it easy for the programmer to write good, clean, and readable code. Having a good collection of structured constructs (like the DO-LOOP and the IF-THEN-ELSE) helps tremendously. Being able to use uppercase or lowercase, blank space, and blank lines also helps. True BASIC meets these tests, as do several other languages. It is easy to write clean code in True BASIC, and it is therefore more likely that clean code will be written!

WHAT ABOUT LOGO?

LOGO is one of the most highly touted languages for schools.[2] Its graphics are especially nice, and students in the lower elementary grades have little trouble learning how to draw simple figures using what is called "turtle graphics." And they quickly learn how to draw complicated figures from simple figures using "recursion." (Recursion is the process whereby subroutines call themselves, either directly or indirectly. Used correctly, recursion can greatly simplify many problems in computing.)

Why are LOGO graphics called "turtle graphics"? The earliest versions of LOGO did not use CRTs. Instead, students typed LOGO commands on a keyboard, and these caused a mechanical "turtle" to move slowly across the floor in the direction and distance given by the student. Nowadays, however, most LOGO systems draw their pictures on CRT screens, with a tiny mark like a triangle acting as the turtle.

Let's look at a simple LOGO application, drawing a square. The student learns to write the following statements:

```
FORWARD 50 RIGHT 90
FORWARD 50 RIGHT 90
```

```
FORWARD 50 RIGHT 90
FORWARD 50 RIGHT 90
```

A little later, the student learns to "condense" this program with a simple looping structure:

```
REPEAT 4 [FORWARD 50 RIGHT 90]
```

The student can also "package" the program using subroutines:

```
TO SQUARE
   REPEAT 4 [SIDE]
END

TO SIDE
   FORWARD 50
   RIGHT 90
END

SQUARE
```

Thus, SQUARE is a subroutine that can be "invoked." And SQUARE in turn invokes another subroutine, SIDE. SQUARE is invoked merely by typing SQUARE.

LOGO graphics are "relative" graphics. That is, each movement of the "turtle" is given in terms of changes from its current position. Now the student quickly learns that the screen has a coordinate system that runs from something like (0,200) in the x-direction and (0,100) in the y-direction. The student usually doesn't know the exact position of the "turtle," but he or she can easily see on the screen approximately where it is.

BASIC graphics, in contrast, are "absolute" graphics. To draw the same picture, a square, the BASIC programmer would have to know that the center of the screen is about (100, 50) and then proceed as follows:

```
PLOT LINES: 100, 50; 100, 100; 150, 100; 150, 50; 100, 50
```

True BASIC also allows large pictures to be built from smaller ones. For example, the same square might be drawn with:

```
DRAW square (50) with shift(100,50)

PICTURE square (s)
      PLOT LINES: 0,0; 0,s; s,s; s,0; 0,0
END PICTURE
```

The PICTURE, which is like a subroutine, draws a square whose side is s, with the lower left-hand corner at the origin. The main program "draws" this square (the side is specified to be 50) after "shifting" the picture to the point (100,50). The square shown on the screen thus has sides of

length 50 and has its lower left-hand corner at the point (100,50).

We don't deny that LOGO is easier for this application. But we have chosen a simple example most favorable for LOGO. True BASIC does better when plotting curves of functions. To plot curves, the LOGO programmer would have to

1. Calculate the differences in both x and y for successive points on the graph.
2. From the differences, calculate the new angle.
3. From that angle, calculate the change in angle. Only then can the "turtle" be moved to the new point.

LOGO is used almost entirely as a plotting language. To give a flavor of a typical nonplotting construct in LOGO, we start with a True BASIC IF-THEN-ELSE construct that finds the minimum "a" of two numbers without using the MIN function.

```
IF x < y then
   LET a = x
ELSE
   LET a = y
END IF
```

LOGO doesn't allow an ELSE part, so we make a slight change to yield this equivalent construct:

```
LET a = y
IF x < y then LET a = x
```

The same construct would appear in LOGO as

```
MAKE "A :Y
IF :X < :Y [MAKE "A :X]
```

The reasons for the unusual punctuation are complicated, and we will not discuss them. The rest is clear. LOGO uses "MAKE" instead of "LET," and the brackets [and] are used to group statements (here there is only one statement inside the brackets, but they are still required).

Besides turtle graphics, LOGO includes extensive facilities for handling "list structures." It is, however, fair to say that LOGO does not readily allow the construction of certain kinds of large programs, especially those that don't use graphics. In fact, most versions cannot even deal with fractional numbers and strings. What LOGO does, it does

very well; namely, it gives beginning students an extremely simple language for learning the concepts of programming, including recursion, through plotting.

WHAT ABOUT PASCAL?

Pascal is the language most widely used for computer science instruction.[3] It is a simple language, and yet allows structured programming. In fact, its popularity is based largely on its having been the first widely used language that is both simple and structured. Since its introduction over a decade ago, many textbooks have been written for it, and one can find a good version of Pascal on practically every kind of computer, big or small.

Believers in Pascal excoriate BASIC for having few structured constructs. But again, they are talking about Street BASIC. True BASIC contains a full set of structured constructs. In fact, it contains more than are found in Pascal.

As examples, let's look again at the loop construct. First, the variation with the test at the beginning. In Pascal, it looks like this:

```
while x < y do begin
        ...;...
end
```

The corresponding construct in True BASIC is

```
DO while x < y
        ...
        ...
LOOP
```

The second example shows the "test" at the end of the loop.

```
repeat
        ...;...
until x >= y
```

The corresponding construct in True BASIC is

```
DO
    ...
    ...
LOOP until x >= y
```

A third variation, when there is a given range of values, is

```
for i := 1 to n do begin
    ...;...
end
```

which is the analogue of the True BASIC

```
FOR i = 1 to n
     ...
     ...
NEXT i
```

So far, True BASIC and Pascal are "neck and neck," although the fact that Pascal uses two completely different constructs for the "test" at the beginning or at the end of the loop is a mystery to many. But True BASIC "pulls ahead" by allowing the "until" on the DO statement, and the "while" on the LOOP statement. And Pascal loses ground by allowing only a step size of 1 in its version of the for-loop.

What allows True BASIC to "win by several lengths" is its middle exit from the loop. To understand the importance of the middle exit, we recall the example of searching for a particular name in a list. In True BASIC this might be done with

```
! The names are in the string list names$ whose
! subscripts run from 1 to n.
FOR i = 1 to n
     IF desiredname$ = name$(i) then EXIT FOR
NEXT i
! If i = n+1, then the desired name was not found.
```

The standard way to program this in Pascal is:

```
(* The names are in the string list names whose
   subscripts run from 1 to n.*)
i := 1;
while i <= n and desiredname <> names(i) do begin
       i = i + 1;
end;
(* If i = n+1, then the desired name was not found. *)
```

We believe that the Pascal coding fragment is harder to read and to understand. There are several reasons.

1. The inside part of the loop contains only the bookkeeping instruction of increasing the value of i.
2. The "good part" is in the test for continuing the loop, but it is backward; that is, the test is <> rather than =.

3. This reason is rather subtle. The two parts of the logical test "i <= n" and "desiredname <> names(i)" must be computed in order; that is, if the first is not met, the second should not even be examined. Why? Because, if i is really > n, the list element "names(i)" doesn't even exist!

Pascal does not guarantee the order in which the two parts of the test are computed. Even if it did, and the above example worked correctly, then the following variation would not work when the desired name is not in the list.

```
(* If the previous example works, this one won't *)
i := 1;
while desiredname <> names(i) and i <= n do begin
      i = i + 1;
end;
```

Some gurus of structured programming insist that there should be only one exit condition from a loop. The above approach meets this condition, sort of. Thus, while admitting to the flaws we cite, they grudgingly prefer the above approach as a lesser evil.

Another way to code the Pascal version uses a goto statement to exit from the loop.

```
      for i := 1 to n do
          if desiredname = name(i) then goto 1234;
1234: . . .
```

This version is perhaps easier to read, but most Pascal textbooks will probably advise against using the goto statement on dogmatic grounds. (Notice that statement labels look like line numbers, but are not.) Instead, they might recommend using flags, as in this example:

```
i := 1; found := false;
while i <= n and not found do begin
      if desiredname = name(i) then found := true;
      i := i + 1
end
```

The trouble is that flags often cause more confusion than goto statements. The conclusion is inescapable—a middle exit from a loop is an important and useful programming construct.

True BASIC also includes a middle exit from a DO loop (the EXIT DO statement), and from subroutines and defined functions (the EXIT SUB and EXIT DEF statements).

In another arena, Pascal fights back with its most important weapon. It allows the user to declare new data structures. For example,

```
employee = record
     name:  array [1..20] of char;
     salary: real;
     age:    integer
end
```

It is obviously useful, in complicated applications, to be able to deal with the entire employee record without having to deal individually with the little pieces of it.

But this capability is a double-edged sword. On the one hand, Pascal allows programmers to define and use complicated data structurns. On the other hand, it requires all programmers to declare *all* variables, even for very simple programs.

Pascal's requirement that all variables be declared gives True BASIC the opening it needs to fight back. True BASIC does not require you to declare variables—they are "declared" from their appearance, the string variables having the final $ and the numerical ones not. In True BASIC the only data structures that can be declared are arrays—lists and tables. They are so declared with the DIM statement.

Beginners therefore do not need to learn about declarations in True BASIC (except for lists and tables), and they can thus learn True BASIC more quickly. And their simple programs are much shorter than Pascal's because they don't have to contain those declaration statements.

Incidentally, Pascal requires that the user know about two kinds of numbers—integer and real. Reals are used for general computation, while integers must be used in for-loops and as array subscripts. Pascal thus permanently burdens the user with matters that are temporary peculiarities of the ways computers do arithmetic.

Another problem with Pascal is its dependence on begin-end pairs to group statements. This is most evident in the loop examples above. In the while-do loop, a single statement must follow the "do." If several statements are to be carried out repetitively, they must be converted into a single statement with a begin-end pair. In contrast, the repeat-until loop allows zero, one, or many statements between the "repeat" and the "until". To stress this point, consider that the following two lines are not equivalent:

```
while i <= n do i := i + 1;
```

```
while i <= n do; i := i + 1;
```

but the following two are equivalent:

```
repeat i := i + 1 until i > n;

repeat; i := i + 1 until i > n;
```

We leave it to the reader to figure this one out.

Pascal suffers even more when it tries to handle strings of characters. True BASIC string variables can contain strings of any length. For example, one time the variable "answer$" might contain "yes" and another time it might contain "no." Such string variables are said to be "dynamic" in that their contents can vary in length.

A Pascal string variable is always an array (list or vector) of characters, and is therefore always exactly as long as the array declaration specifies. This can be seen from the data-structure declaration shown above. The part called "name" is an array of length 20 (the subscripts run from 1 to 20), always. If the employee's name is Jones, which contains only five characters, the rest of "name" must be filled with spaces. Furthermore, because Pascal is so strict about "type matching," the following is simply not allowed.

```
name := "Jones"
```

Naturally, True BASIC (and most other versions of BASIC as well) do allow

```
LET name$ = "Jones"
```

regardless of length.

Our next comparison between True BASIC and Pascal concerns parameters. Parameters are sometimes called arguments, and appear within parentheses in True BASIC SUB and CALL statements. Parameter passing is a subtle and complicated subject, and a full discussion is beyond our purposes. But Pascal's strict typing philosophy when applied to parameters may be carrying things too far. We start with a simple True BASIC subroutine that adds the elements of a list from 1 to n.

```
SUB add (list( ), n, sum)
    LET sum = 0
    FOR i = 1 to n
        LET sum = sum + list(i)
    NEXT i
END SUB
```

This subroutine could be used by calling it from somewhere else in the program, as follows:

```
DIM x(25)
  . . .
CALL add (x, 10, s)     ! We don't use all entries
```

In this part of the program, the list is named x, we want the sum of the entries from 1 to 10, and the resulting sum will be "returned" in the variable s.

Pascal requires that the parameters in the "procedure heading" must match, exactly, the parameters in the "activation" of that procedure. The same sequence in Pascal might look something like this:

```
procedure (list : array [1..100] of real,
           n:     integer,
           sum : real);
begin
      sum := 0.0;
      for i := 1 to n do sum = sum + list[i]
end;

(* Now we come to the main program *)

var x : array [1..100] of real;
    s : real;
...
add (x, 10, s);     (* This is the activation *)
```

Notice that x, the array in the activation, contains exactly 100 elements, and that "list," the corresponding parameter in the procedure-heading, also contains exactly 100 elements. "As long as the program works okay," you might ask, "what's the problem?"

The problem is precisely this: if you wanted to add up numbers in a list having size 50, as shown below, you simply could not use the above procedure. Pascal wouldn't allow you to "match" a list of size 50 with a list of size 100:

```
(* Assume the same procedure as above *)
var y : array [1..50] of real;
    s : real;
          ...
add (y, 10, s);
```

Of course, you could rewrite the "add" procedure. Or, you could have a procedure "add100" that adds lists of length 100, a procedure "add99" that adds lists of length 99, and so on. But this is foolish. The whole idea of subroutines is that you write one to do some general task, and then use it in a variety of situations. Pascal's strict typing requirement gets in the way when arrays are used. Incidentally, this nonsense also applies to strings, since strings are arrays (lists) of characters in Pascal.

True, a few versions of Pascal in common use depart from the rigid strictures of ISO Pascal and allow what are called "conformant arrays," which is a fancy name for what True BASIC does automatically. But why

is it necessary to make an international incident out of something that ought to be included without any fuss? (ISO stands for the International Standards Organization.)

Other features that are simpler to work with in True BASIC deal with vectors and matrices in the MAT statements (these are virtually missing in Pascal), and graphics (which are completely absent in Pascal). In addition, Pascal offers only one way to print numbers without specific formatting (it uses the e-format, even when the number happens to be a small integer), provides no standard way to open files or manage libraries, and contains no mechanism for detecting exceptions. (The e-format is the one that prints "1.00000e+2" when the number is 100. An exception is an error that occurs after the program has started running, such as, for example, attempting to divide by 0.)

WHAT ABOUT FORTRAN?

The best we can say about FORTRAN (or COBOL, for that matter) is this: if you must learn it for your job, fine and good; otherwise, ignore it for as long as possible.

FORTRAN was the first widely used programming language, invented in the prehistoric 1950s. The language has been extended and partially cleaned up since then, but it remains basically ugly and is simply not a good language for the beginning programmer.

To illustrate just one ugly feature, let's look at a typical DO loop in FORTRAN.

```
      DO 10 I = 1,3
         N = N + I
10 CONTINUE
```

This is, of course, equivalent to the True BASIC

```
FOR i = 1 to 3
    LET n = n + i
NEXT i
```

I is sobering to realize that typing a single wrong character can cause that FORTRAN loop to be changed completely, while still remaining legal. In fact, just such a single typing error caused the destruction of an unmanned satellite several years ago—the programmer accidentally typed a period instead of a comma in the first line.

```
      DO 10 I = 1.3
         N = N + I
   10 CONTINUE
```

The reason for the confusion is that FORTRAN, like old-fashioned BASICs, ignores spaces. This last version thus was transformed to

```
      DO10I=1.3
      N=N+I
   10 CONTINUE
```

which is equivalent to the True BASIC

```
LET do10i = 1.3
LET n = n + i
```

The DO statement was thus accidentally turned into an assignment (LET) statement.

Now this silly, but astronomically expensive, error could not occur in True BASIC for three reasons:

1. True BASIC uses keywords (like "to") to separate the parts of a loop, instead of punctuation, which is hard to remember and easy to misuse.
2. True BASIC does not allow spaces within variable names. A two- or three-word sequence could never be confused with a single word containing the same letters without spaces.
3. True BASIC would complain about the absence of a LET keyword.

FORTRAN, for good historical reasons, brings the programmer closer to the machine hardware. But these same reasons render it less beneficial for the beginning programmer. We give but one example.

When FORTRAN was invented in the 1950s, computers were expensive and slow. It was therefore of paramount interest that FORTRAN be as efficient as possible; that is, programs should run as fast as possible. It was decided to save time by not checking to see that subscripts stayed in bounds. Thus, the program

```
      DIMENSION X(10)
      DO 10 I = 1,100
         X(I) = 123.456
   10 CONTINUE
```

would store the number 123.456 in 100 different memory locations, only 10 of which belonged to the list X. Who can guess what damage might be done? And there would be no error message to warn the programmer.

The corresponding program fragment in True BASIC

```
DIM X(10)
FOR i = 1 to 100
    LET X(i) = 123.456
NEXT i
```

would produce a "subscript out of bounds" error message as soon as the variable i equaled 11.

Since FORTRAN was invented, computers have become so fast that most of us are glad that a language like True BASIC always spends a little time checking for subscript errors.

WHAT ABOUT COBOL?

COBOL was designed in the 1960s, mainly for business programming. It is extremely verbose, with many long keywords. There is no such thing as a simple program in COBOL. Most of the language deals with input and output of data records from files. Just about the most complicated arithmetic it is ever called upon to do involves adding numbers and calculating percentages. It cannot even take a square root! COBOL is simply not a good language for the beginning programmer. It has been suggested that COBOL is also not a good language for the business programmer, but that is another matter.

WHAT ABOUT Ada®?

Ada, the new Defense Department language, contains just about anything anyone ever thought of.[4] Of course, it is a structured language. But it is much more, although it is way beyond the purposes of this small book to describe it in detail. Suffice it to say that Ada is so large that you are not likely to find true Ada compilers or interpreters on small personal computers for some time to come.

*Ada is a registered trademark of the U.S. Government–Ada Joint Program Office.

The size of Ada as a language means that learning it is a monumental task for any programmer, especially a beginning programmer. But the elementary programming concepts used in Ada differ little from those in other languages. True BASIC thus provides a good introduction for the aspiring Ada programmer.

What is surprising about Ada is that its structured constructs bear a striking resemblance to BASIC's. Let's examine the simple program fragment we used earlier, the one that finds the minimum of two numbers (without using the MIN function).

```
IF x < y then
   LET a = x
ELSE
   LET a = y
END IF
```

The corresponding fragment written in Ada would look like this:

```
if x < y then
  a := x
else
  a := y
endif
```

One must use a magnifying glass to spot the differences (the choice of uppercase or lowercase letters doesn't matter in either language). First, the assignment operation (LET statement in BASIC) is slightly different; in Ada no keyword is used, and the symbol is := instead of =. Second, the "endif" in Ada is spelled without a space.

COMPARISONS IN BRIEF

In the following, we will briefly compare and contrast the languages we have been discussing. Most of the comparisons are made without further comment. Nor will we explain the reasons for the choices made in the other languages.

Assignment Statements

The first feature we compare is the assignment statement. You probably know it as the LET statement.

BASIC	`LET x = 3`
LOGO	`MAKE "X3`
Pascal	`x := 3;`
FORTRAN	`X = 3`
COBOL	`ASSIGN 3 TO X`
Ada	`x := 3;`

FORTRAN is confusing because "X = 3" can look like a "test," as in

```
IF X = 3 THEN . . .
```

COBOL is verbose because it was once thought that anyone in business couldn't possibly understand it otherwise.

Structured Constructs

We touched earlier on structured constructs and their use. (See chapter 4 and additional examples in chapter 7.) We simply note here that True BASIC, Pascal, and Ada are well equipped with structured constructs (although the details vary), while FORTRAN and COBOL are not (although FORTRAN-77 has some). LOGO has few structured constructs, since it depends on recursion to achieve its purposes.

More about Declarations

Declarations are statements in the program that do no computation but do state certain facts about the program's variables and other quantities. True BASIC uses declarations only for giving the dimensions of arrays (vectors and matrices), and for asserting the existence of functions and subroutines that the compiler or interpreter may not yet have seen. String variables are distinguished from numeric variables by having their names end with a dollar sign ($). Strings and numbers are the only types of variables in True BASIC, so this convention works quite well.

FORTRAN does not require declarations in all cases, and is thus similar to BASIC. FORTRAN does permit the programmer to declare all variables if he or she wishes. In fact, declarations can be used to override the old I, J, K, L, M, or N convention for integer-valued variables. In addition, FORTRAN includes double-precision and complex number types.

In COBOL all variables are "typed" according to the format to be used when printing their values. By "format" we mean the number of digits, the number of decimal places, whether a $ is present or not, etc.

Pascal and Ada are "strongly typed" languages in that all variables and other quantities must appear in some declaration statement. As we have stated, for the beginner it is an advantage not to have to declare all variables. BASIC's lack is therefore its advantage. Beginning programmers do not understand why they have to declare all their variables when all they are doing is simple arithmetic or simple string processing.

Data Structures

The only data structures allowed in BASIC are arrays, which are collections of numbers or of strings, selected with subscripts. More general data structures are not included.

Like BASIC, neither LOGO nor FORTRAN includes more general data structures, although LOGO supports a sophisticated application called list processing.

COBOL requires that the format (how many digits, how many decimal places) of all variables be declared.

Pascal and Ada allow declaration of almost any data structure you can imagine. This is at once their main strength and a source of their complexity.

Subroutines

True BASIC includes internal and external functions and subroutines. That is, the code for a subroutine can be inside (internal to) the main program, or it can be outside (external to) the main program. For example, subroutines in libraries are external. The same goes for user-defined functions, whether they be of the single-line or multiline variety.

LOGO allows only for internal subroutines.

Pascal includes internal functions and subroutines. Some versions of Pascal, but not ISO Standard Pascal, allow for external functions and subroutines as well.

Fortran has internal one-line function definitions and external functions and subroutines, but lacks internal subroutines.

COBOL has external subroutines, but lacks true internal subroutines. Instead, programmers are forced to use the PERFORM command, which does not allow arguments.

MAT Functions

True BASIC allows one to deal with vectors and matrices as entities. As an example, suppose we have a list (vector) of wholesale prices of items and a corresponding list of retail prices, and we want a list of the profit margins. One way to accomplish this might be

```
DIM wholesale(100), retail(100), margin(100)
    ...
FOR i = 1 to 100
    LET margin(i) = retail(i) - wholesale(i)
NEXT i
```

Much more succinct, and more readable, is this version:

```
DIM wholesale(100), retail(100), margin(100)
    ...
MAT margin = retail - wholesale
```

Of all the languages we are discussing, only Ada can deal with entire arrays in this way, but the programmer himself must "declare" what he means by the subtraction of two vectors. These and other matrix commands are included automatically in True BASIC.

Graphics

BASIC has a complete graphics package containing a wide range of monochrome and color two-dimensional graphics, including inputs from various devices such as "mice."

LOGO is built around its famous "turtle" graphics. Turtle graphics are "relative" in that the new position is determined relative to the current position. Most of BASIC graphics is "absolute" in that one draws lines, etc., between given points.

Pascal, FORTRAN, COBOL, and Ada contain no graphics, although some personal computer versions of Pascal do have them.

TRUE BASIC WINS GOING AWAY

We feel that True BASIC is just about the ideal language for learning programming. It doesn't matter if the goal is to understand computers, to

support courses in mathematics and science, or to prepare for a career in programming. True BASIC is easy to learn and yet contains almost all of the concepts you will ever encounter in programming—subroutines, structured constructs, simple operations with files, and so on.

If your ultimate purpose is to learn Pascal, FORTRAN, COBOL, or Ada, you will be speeded on your way by starting with True BASIC. All you must do is learn the new grammatical rules, and possibly data structuring. True BASIC will have already taught you the fundamental concepts.

And if you are not required to learn another language, you will be quite happy with True BASIC. It contains many of the features found in other languages. In addition, True Basic contains features not found in other languages, like fancy graphics and MAT statements. And True BASIC is rich enough to allow you to construct large programs.

True BASIC nicely bridges the gap between LOGO, which is widely used in the lower grades, and Pascal and Ada.

7
FURTHER EXAMPLES

Let us now illustrate the wide variety of applications that may be written in True BASIC. We have selected examples that are interesting in themselves, and together they illustrate a variety of features.

The examples are independent of each other. Therefore, the reader may pick and choose among them. While a few are mathematical examples, most require no knowledge of mathematics.

In each case we show a listing of the program, supply commentary on it, and show the output of a "run" of the program.

The first example calculates a value for pi, the number fundamental to geometry and to many other branches of mathematics. This number cannot be expressed as a finite decimal expansion. Hence our goal is to calculate its value to a reasonable number of places. We will use one of the simplest ways of calculating pi. As you will see from the run, the answer is correct to five decimal places after just nine iterations!

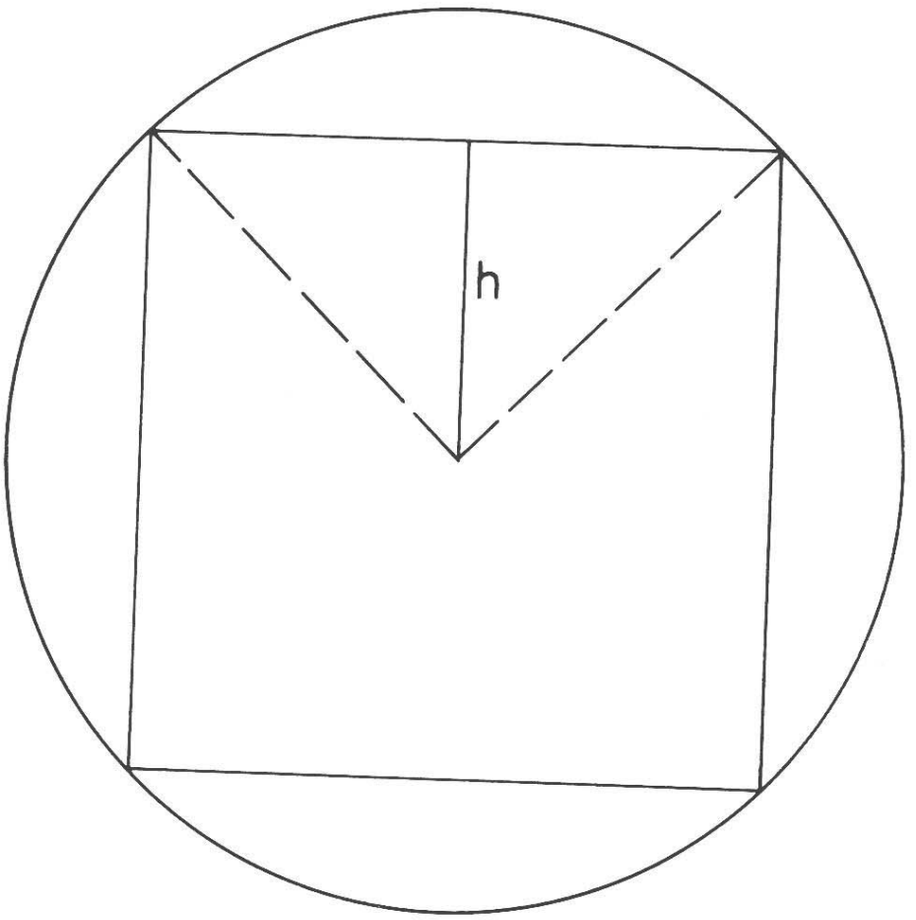
The method is based on some simple facts from high school geometry: pi is equal to the area of a circle whose radius is 1. We may approximate this area by calculating the area of a regular polygon. We start with a square inscribed in the circle. Then

```
a = area = 2
```

And if we divide the square into triangles (with vertex at the center of the circle, Figure 7.1) the altitude of each triangle is

```
h = altitude = 1/sqr(2)
```

FIGURE 7.1

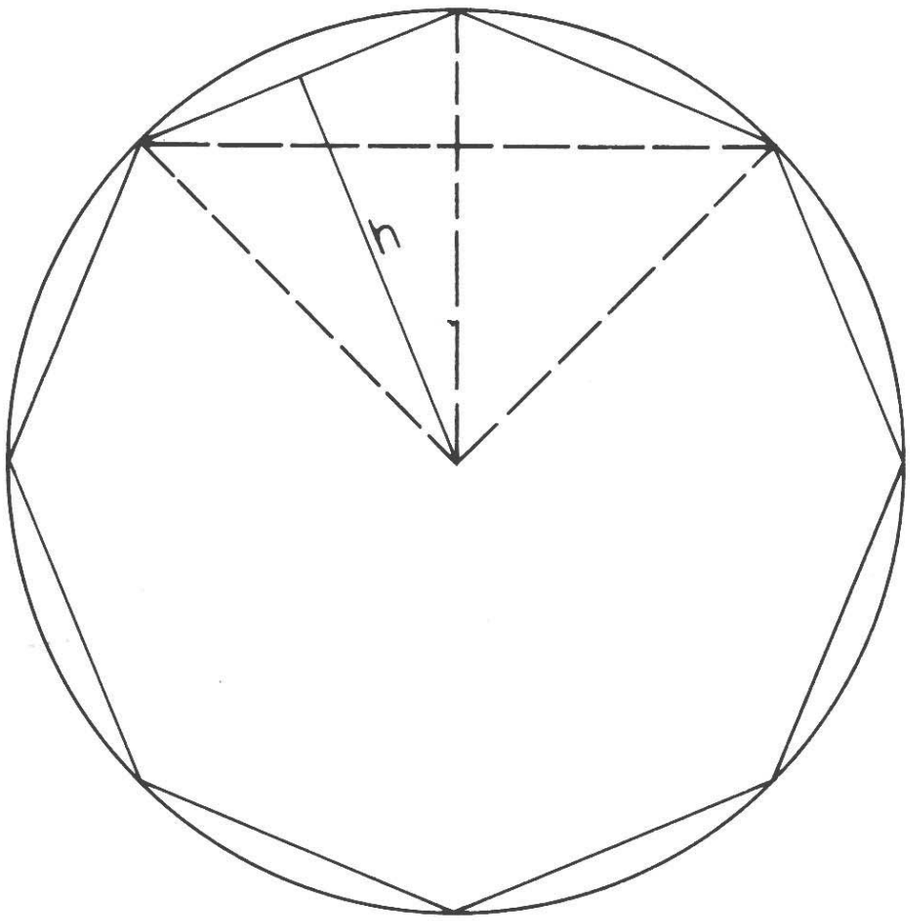


Each time we double the number of sides, we get a much closer approximation of the area of the circle (Figure 7.2). And from calculations with right triangles we can show that

```
new a = a/h

new h = sqr((1+h)/2)
```

The program incorporates these simple formulas. The key structure is the FOR-NEXT loop. Each iteration corresponds to another doubling of the number of sides.

FIGURE 7.2




```
PI

REM Find pi

LET a = 2                          ! Area
LET h = 1/sqr(2)                   ! altitude
FOR k = 1 to 10                    ! Double sides
     LET a = a/h
     LET h = sqr((1+h)/2)
     PRINT a
NEXT k

END
```

```
run

 2.82843
 3.06147
 3.12145
 3.13655
 3.14033
 3.14128
 3.14151
 3.14157
 3.14159
 3.14159
```

The next program calculates the prime numbers up to 200. Prime numbers have fascinated both mathematicians and amateurs over the centuries. Primes are integers, starting with 2, which are not divisible by smaller integers (other than 1).

Primes may be calculated by a simple method known as a "sieve." We start with 2, which is a prime. Then we cross out its multiples, since these are not primes. We take the first number not crossed out, which must be a prime. And we cross out its multiples. And so on.

For example, start with these numbers:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25
```

The number 2 is a prime. Cross out its multiples:

```
3 5 7 9 11 13 15 17 19 21 23 25
```

Now we know that 3 is prime. Cross out its multiples:

```
5 7 11 13 17 19 23 25
```

Next, 5 is a prime. The only multiple (other than 5 itself) is 25:

```
7 11 13 17 19 23
```

Then 7 is a prime. And so on.

In the program we use the list "slot" to represent the integers. At the start, all entries are 0, hence we let 0 stand for "not crossed out." We cross out the number n by letting slot(n) = 1.

You may have noticed in the above example that there were no multiples of 7 left to cross out. The reason is that, if 7 divides a number up to 25, the quotient must be less than 7. Hence the number was already crossed out. For example, 21 = 3*7 is crossed out by 3. Thus the six numbers remaining above are all primes. This feature is incorporated into our program to speed it up. We need only cross out with numbers p such that p*p is no greater than 200.

This is a good exmple of the use of IF structures. It wasn't essential to make "sieve" a subroutine, but such segmenting makes the program easier to read. And we used the variable "upper" to stand for 200. If we want the primes up to 400, we have to change only one line, rather than hunt for all places where 200 occurs. If we wanted to go up above 500, we would also have to change the DIM, and make sure that our computer has sufficient memory.

```
PRIMES

REM Find primes to 200

DIM slot(500)                    ! For crossing out
LET upper = 200                  ! How far to go

FOR p = 2 to upper
     IF slot(p) = 0 then         ! Prime
        PRINT p;
        IF p*p <= upper then CALL sieve
     END IF
NEXT p

SUB sieve                        ! Cross out multiples of p
     FOR i = 2*p to upper step p
          LET slot(i) = 1
     NEXT i
END SUB

END

run

2 3 5 7 11 13 17 19 23 29 31 37 41 43
47 53 59 61 67 71 73 79 83 89 97 101 103
107 109 113 127 131 137 139 149 151 157 163
167 173 179 181 191 193 197 199
```

For a change of pace, let's examine a program that counts the number of words in a file. It asks the user for the name of the file, reads the lines of the file one at a time, counts the number of words, and prints the total. We used the skeleton of this program earlier to illustrate how subroutines help us to segment a program.

Before using a file, BASIC must open it. That means that it must ask the operating system for access to the file. The statement "OPEN #1:" asks for such access, and if access is obtained, the file can be referred to as #1. The subroutine "getfile" asks for the name and opens the file.

The subroutine "words" counts the words on the current line. The

variable p is a pointer, which moves along the characters of the line. Words are identified by the fact that they are followed by a space; thus the function "pos" helps us to find the end of the word.

There are two complications, however. First of all, there may be more than one space after a word. For example, the end of a sentence is commonly followed by two spaces. Therefore, "words" calls "blanks" to move the pointer past any blanks and hence to the next word. And there is usually no space at the end of the line. This could be accounted for by a special end-of-line check. We will use the simpler trick of attaching a space to the end of each line, so that all words will end with a blank.

The run counts the number of words in a draft of chapter 4.

```
WORDS

REM Count words in a file

CALL getfile                          ! Program skeleton
LET count = 0
DO while more #1
   LINE INPUT #1: line$
   CALL words
LOOP
PRINT "Number of words:"; count

SUB getfile                           ! Open file
    PRINT "Name of file";
    INPUT file$
    OPEN #1: name file$
END SUB

SUB words                             ! Words on a line
    LET line$ = line$ & " "
    LET p = 1
    DO
       CALL blanks
       LET p = pos(line$, " ",p)
       IF p=0 then EXIT DO ! Middle exit
       LET count = count + 1
    LOOP
END SUB

SUB blanks                            ! Move past blanks
    DO while line$[p:p] = " "
       LET p = p+1
    LOOP
END SUB

END
```

```
run

Name of file? chap4
Number of words: 4487
```

One of the powerful features of BASIC is the set of MAT instructions. They allow manipulating arrays by single instructions, eliminating the need for loops and double loops. We will illustrate their use by writing a simple program to average grades.

We allow for a class to have up to 50 students and each student to have up to 10 grades. We will compute a "weighted average" of the grades of each student. The grades will be stored in a table "grades," the weights in the list "weight," the computed averages will go into "average", and we store the names in "name$." The first three of these hold numbers, the last one strings (names).

The information is stored in the DATA statements. We first read the actual number of students and of grades. Then a single MAT statement allows us to read all the remaining information. For example,

```
MAT READ grades(s,g)
```

will read a table of s rows and g columns. One must only be careful that the order of the data is the same as the order of the READ statements.

Next, a single MAT statement computes all the averages. Finally, we print a tabulation of names and averages.

For a very large class it is usually more convenient to store the data in a file. One would then MAT READ from the file.

Even such a simple program illustrates the convenience of MAT instructions. All the common matrix operations, including inverting of a matrix, may be carried out with simple MAT instructions.

```
GRADES

REM Averaging grades

! Allow up to 50 students and up to 10 grades per student
DIM grades(50,10), weights(10), average(50), name$(50)

READ s, g                          ! Number of students, grades
MAT READ grades(s,g), weights(g), name$(s)   ! Read data
MAT average = grades * weights     ! Average grades

FOR i = 1 to s                     ! Print for each student
    PRINT name$(i), average(i)
NEXT i

DATA 6, 4                          ! 6 students, 4 grades
DATA 87, 78, 82, 86                ! Grades
DATA 87, 99, 82, 100
```

```
DATA 73, 82, 67, 85
DATA 59, 62, 73, 81
DATA 88, 88, 88, 88
DATA 77, 61, 59, 93
DATA .2, .2, .2, .4                    ! Weights
DATA Abel, Baker, Cross, David, Elmer, Forest   !Names

END

run

Abel            83.8
Baker           93.6
Cross           78.4
David           71.2
Elmer           88
Forest          76.6
```

Our next example is a game. Two players take turns thinking of a number from 1 to 100. The other player takes guesses and is told whether the guess is too small or too large. The player needing fewer guesses wins.

This is our longest example. But the skeleton of the program is short and simple. It would be suitable for a variety of games where the player and the computer take turns, and after each game the player is given the option for another game.

As usual, the skeleton is fleshed out by means of subroutines. They are simplified by means of the defined function "pick(a,b)" which picks an integer at random from a, a+1, a+2, . . . , b.

The subroutine "player" starts with the computer picking a number and then looping through player guesses. Each guess is compared with "number," and an appropriate response is given. The number of guesses is kept track of in "p_guess."

The subroutine "machine" is more complex, since it contains the computer's strategy in guessing. It is made easier to read by putting portions of the code into the subroutines "guessing" and "answer."

The task is to narrow the possible choices. The variables "low" and "high" keep track of what the lowest and highest possible answers can be. One could then pick a number at random between "low" and "high," but that would be poor strategy. Always picking the midpoint would be a very good strategy, but it leads to a dull (very predictable) game. So we compromise and pick a number at random from the middle third of the interval. The number of guesses is kept in "m_guess."

The strategy is carried out in the routine "guessing." Its DO loop in turn calls the routine "answer." This is good structured programming,

even though each subroutine is called only once. If the entire code were shown in "machine," it would be very difficult to read the program.

An important safeguard is built into this routine. If the player makes a single mistake in answering, it is impossible to guess the number. Therefore, we have an outside DO loop, allowing us to go through the guessing sequence more than once. If high < low , then an error has occurred, and we loop back.

The subroutines "instruct," "who_won," and "more" should be self-explanatory. We show one game. The computer was unusually lucky this time!

GUESS

```
REM Number guessing game

CALL instruct                ! Skeleton of program
RANDOMIZE
DO
   CALL player
   CALL machine
   CALL who_won
   CALL more
LOOP until ans$ = "no"   ! End of skeleton

DEF pick(a,b)=int((b-a+1)*rnd+a) ! Integer from a to b
SUB player                   ! Player will guess
    LET number = pick(1,100)    ! Pick number from 1
                                                to 100
    LET p_guess = 0      ! Count guesses
    PRINT "I have thought of a number."

    DO
       PRINT "Guess";  ! Get guess
       INPUT x
       IF x > number then     ! Give hint
          PRINT "Smaller"
       ELSEIF x < number then
          PRINT "Larger"
       ELSE
          PRINT "Correct!"
       END IF
       LET p_guess = p_guess + 1
    LOOP until x = number

    PRINT "You used"; p_guess; "guesses."
    PRINT
END SUB
```

```
SUB who_won                    ! Announce winner
    IF p_guess < m_guess then
       PRINT "You win ***"
    ELSEIF p_guess > m_guess then
       PRINT "I win ***"
    ELSE
       PRINT "It is a tie."
    END IF
    PRINT
END SUB
!
SUB machine                    ! Machine to guess
    PRINT "Your turn to think of a number."
    DO                         ! Repeat in case of error
       DO                      ! Give time to think
          PRINT "Ready";
          INPUT ans$
       LOOP until lcase$(ans$) = "yes"

       LET m_guess = 0         ! Count guesses
       LET low = 1             ! Range in which to guess
       LET high = 100

       CALL guessing           ! Actual guessing

    LOOP until error = 0
    PRINT "I needed"; m_guess; "guesses."
    PRINT
END SUB
```

```
SUB guessing                    ! Machine guess routine
    DO                          ! Start guessing
       LET lowguess = (2*low + high)/3
       LET highguess = (2*high + low)/3
       LET guess = pick(lowguess,highguess)  ! Near
                                                middle
       PRINT "My guess is"; guess
       CALL answer      ! How is it?
       LET m_guess = m_guess + 1
     LOOP until ans$ = "cor" or high<low  ! Correct
                                            guess or error

    IF high<low then
       PRINT "That cannot be."
       PRINT "Let's try again."
       PRINT
       LET error = 1
    ELSE
       LET error = 0
    END IF
END SUB
!
SUB answer                      ! Find out how guess is
    DO                          ! Until get ok answer
        LET ok = 1              ! Error flag
        PRINT "Hint";
        INPUT ans$
        LET ans$ = lcase$(ans$[1:3])  ! Lower case, 3
                                        letters
        SELECT CASE ans$
        CASE "lar"          ! Too small
             LET low = guess + 1
        CASE "sma"          ! Too large
             LET high = guess - 1
        CASE "cor"          ! Correct
        CASE else
             PRINT "Please answer:"
             PRINT "'smaller', 'larger', or 'correct'."
             LET ok = 0
        END SELECT
     LOOP until ok = 1
 END SUB
```

```
SUB instruct                    ! Give instructions
    PRINT "We will take turns thinking of a number."
    PRINT "It must be an integer from 1 to 100."
    PRINT "The other player must guess the number."
    PRINT "After each guess a hint is given:";
    PRINT "'Smaller', 'Larger', or 'Correct'."
    PRINT "The player needing fewer guesses wins."
    PRINT
END SUB

SUB more                        ! Another game$
    PRINT
    DO
        PRINT "Another game";
        INPUT ans$
        LET ans$ = lcase$(ans$)
        IF ans$ = "yes" or ans$ = "no" then EXIT DO
        PRINT "Please answer 'yes' or 'no'."
    LOOP
    PRINT
END SUB

END
```

```
run

We will take turns thinking of a number.
It must be an integer from 1 to 100.
The other player must guess the number.
After each guess a hint is given: 'Smaller', 'Larger', or
'Correct'.

The player needing fewer guesses wins.
I have thought of a number.
Guess? 50
Larger
Guess? 80
Larger
Guess? 90
Smaller
Guess? 85
Larger
Guess? 87
Larger
Guess? 88
Correct!
You used 6 guesses.
```

```
Your turn to think of a number.
My guess is 54
Hint? smaller
My guess is 26
Hint? larger
My guess is 40
Hint? smaller
My guess is 33
Hint? correct
I needed 4 guesses.

I win ***

Another game? no

Ready
```

True BASIC has built into it the capability of drawing graphs and pictures. We will give four examples. Figures 7.3–7.6 are photographs of the output.

We first consider the problem of where the graphs of the sine curve and of the common logarithm intersect. There is no formula available to give us the answer, but we can draw the graphs on the same screen and observe where they intersect.

The program is quite straightforward. SET WINDOW specifies what portion of the plane we wish to see. We set it for x from 0 to 10, and y from −1 to 1. Then we clear the screen and draw the x-axis and y-axis.

The loops for plotting are similar to loops we would use to print tables of values. But instead of a PRINT statement, we use PLOT. Only the semicolon at the end of the line needs explanation: without it points would be plotted. With the semicolon the drawing beam is left on, and hence the points are connected. The PLOT statement with nothing after it turns the beam off, so we can start a new graph.

The SET COLOR statements allow us to draw axes and graphs in different colors.

```
GRAPH

REM Intersection of two curves

SET WINDOW 0,10,-1,1          ! Coordinates
CLEAR

SET COLOR "green"
PLOT 0,0; 10,0                ! x-axis
PLOT 0,-1; 0,1                ! y-axis
```

```
SET COLOR "yellow"
FOR x = 0 to 10 step .2          ! First curve
     PLOT x, sin(x);
NEXT x
PLOT

SET COLOR "red"
FOR x = 1 to 10 step .2          ! Second curve
     PLOT x, log10(x);
NEXT x

END
```

From Figure 7.3 we see that there are three intersections. We will later show a program that finds good numerical values for these points.

Next we will write a program to draw a bar graph or histogram. The information is in DATA statements, starting with how many numbers there are to be plotted and how high the highest value can be. We use these numbers in a SET WINDOW statement, thus automatically achieving the necessary scaling.

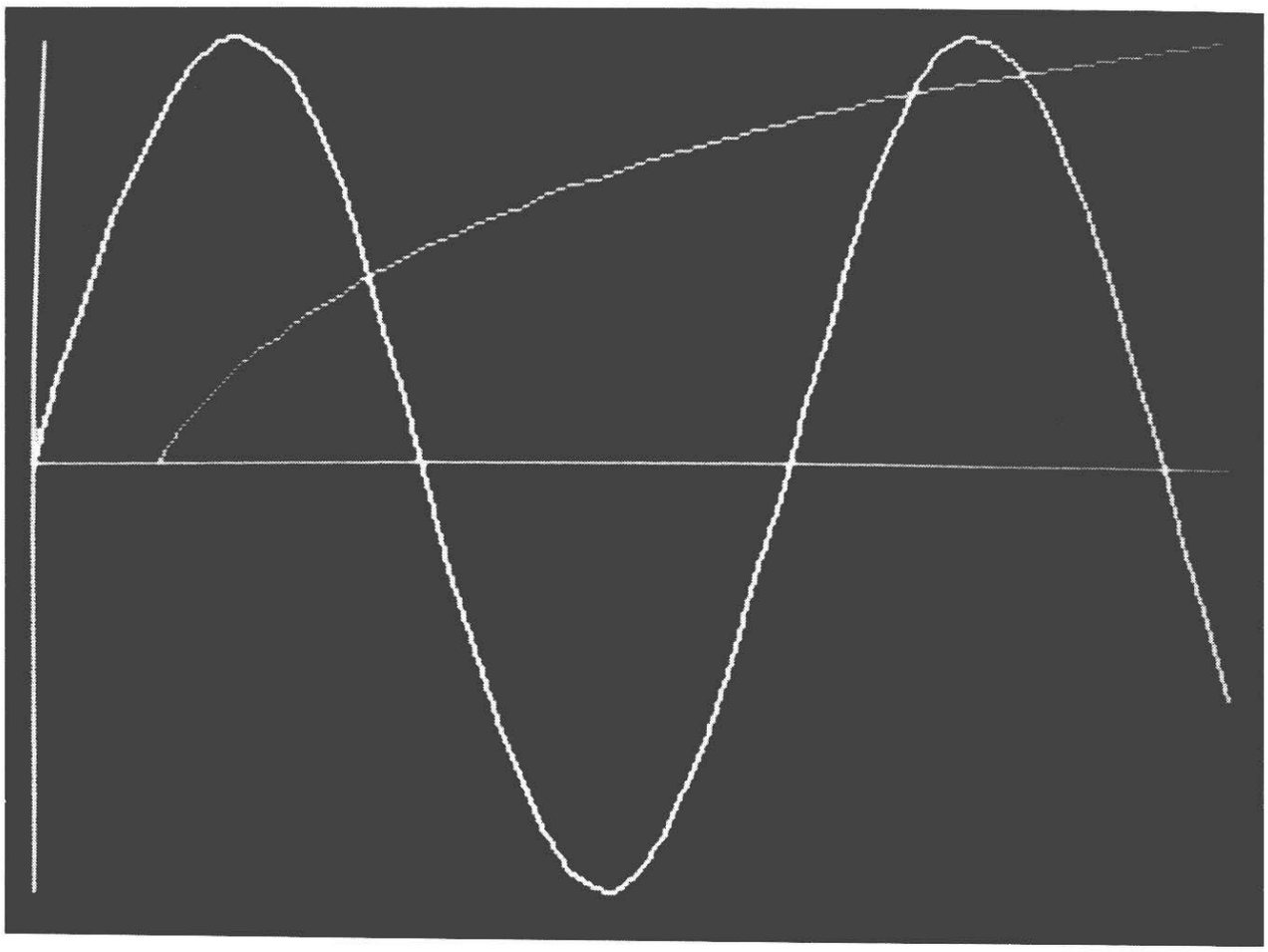
We will plot a horizontal line on which the bars will be drawn. We switch color, to red, for drawing bars, read the numbers one at a time, and draw a bar for each.

This program shows a new feature. A PICTURE is used to define a typical "bar." We then use it with a DRAW command.

A PICTURE is similar to a SUB, but it allows us to apply transformations to the picture. We will discuss PICTURE further in the next example.

```
BAR

REM Draw bar graph

READ n, datamax                  ! How many, how high

SET WINDOW 0,n,0,datamax
CLEAR

PLOT 0,0; n, 0                   ! Axis
SET COLOR "red"                  ! Red bars
FOR i = 1 to n                   ! Bars
     READ number
     DRAW bar(i,number)
NEXT i

PICTURE bar(x,y)                 ! Typical bar
     PLOT x-1,0; x-1,y; x,y; x,0
END PICTURE
```

FIGURE 7.3

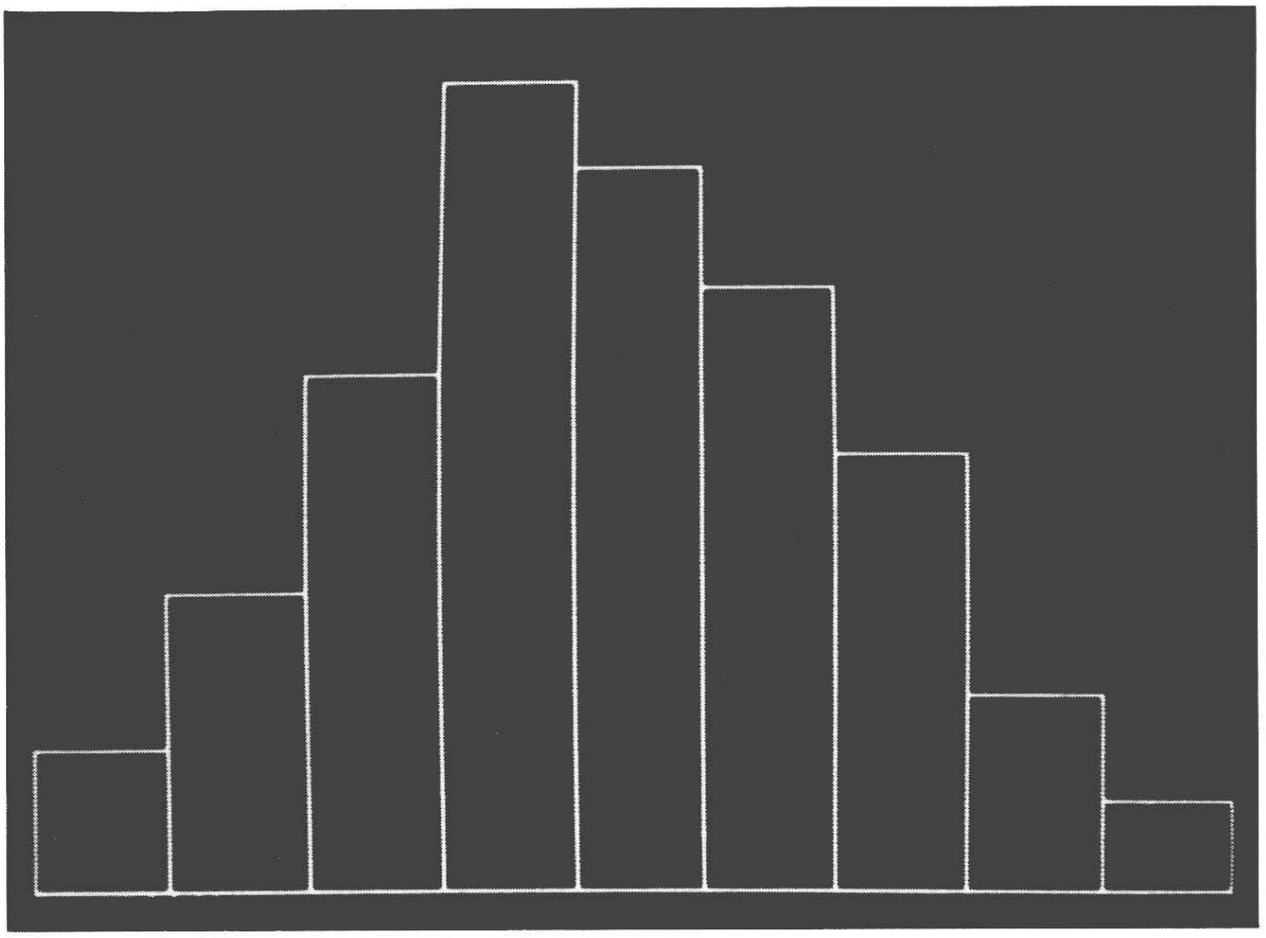


```
DATA 9, 50                    ! No. of data, largest allowed
DATA 8, 17, 30, 47, 42, 35, 25, 11, 5
END
```

As a more interesting use of pictures, we consider the problem of drawing a number of stars in some attractive arrangement. Since a five-sided star is a reasonably complicated shape, it pays to construct a PICTURE for a prototype star. We can later worry about the placement of stars.

We can visualize the star as it is drawn inside a unit circle. The first corner is on "top" of the circle, the next one two-fifths of the way around the circle, and so on. Angles are normally measured in radians in BASIC, but we use the OPTION statement to switch to degrees. So we start at 90 degrees, and keep changing by two-fifths of a whole circle, or 2/5 * 360 degrees. The combination of cosine and sine picks out a point on the unit circle.

FIGURE 7.4



We choose convenient coordinates by means of SET WINDOW and will place the centers of the stars at grid points. The power of PICTURE is shown in the DRAW statement. We tell BASIC to draw a star, at one-half full scale, and have the origin shifted to the point x, y. The actual coordinates of the points drawn are calculated automatically for us!

To make the picture more colorful, colors are picked at random. The IBM PC allows us to choose from a palette of only three colors. On some other computer we might draw stars in a wide variety of colors.

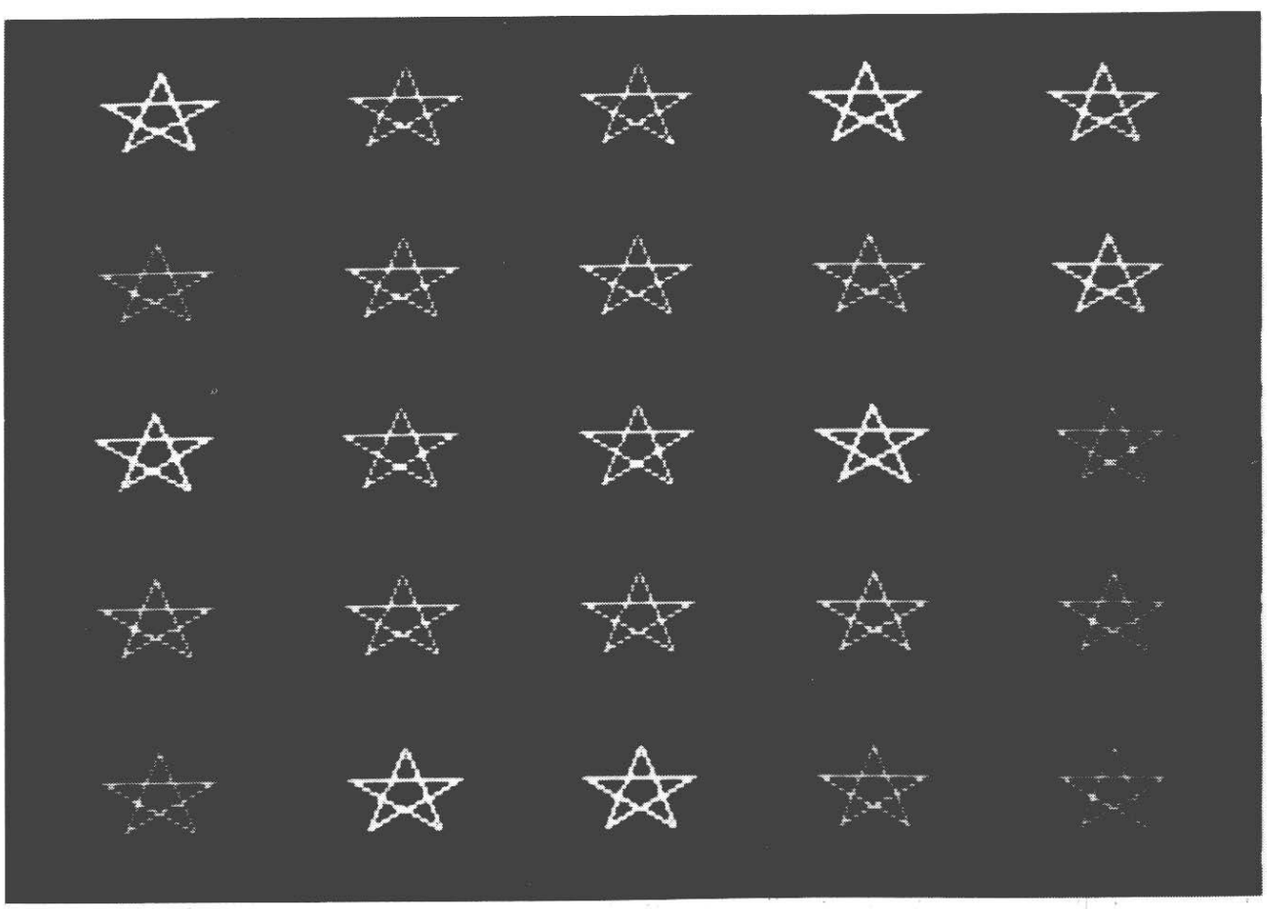
```
STARS

REM Draw stars

OPTION angle degrees

PICTURE star                          ! How to draw a star
     LET new = 90
     FOR i = 1 to 5
          LET old = new
          LET new = old + 2/5*360
          PLOT cos(old), sin(old); cos(new), sin(new)
     NEXT i
END PICTURE
```

```
SET WINDOW 0,10,0,10
CLEAR
FOR x = 1 to 9 step 2     ! Place at alternate grid points
    FOR y = 1 to 9 step 2
        SET COLOR int(3*rnd + 1)     ! Random choice of
                                                color
        DRAW star with scale(1/2) * shift(x,y)
    NEXT y
NEXT x
END
```

Let us now put together the various graphics tools illustrated so far to draw a United States flag. We choose coordinates as follows: thirteen units vertically, for the thirteen stripes, twenty-seven horizontally to have roughly the right proportion for the flag.

First we draw the thirteen bars. That is very easy with the BOX AREA statement. (It takes coordinates in the manner of the SET WINDOW statement, and draws a rectangle of a solid color.) But we must alternate between red and white stripes. And the upper stripes will be shorter, to allow for the blue corner with the stars. We use the "mod"

FIGURE 7.5



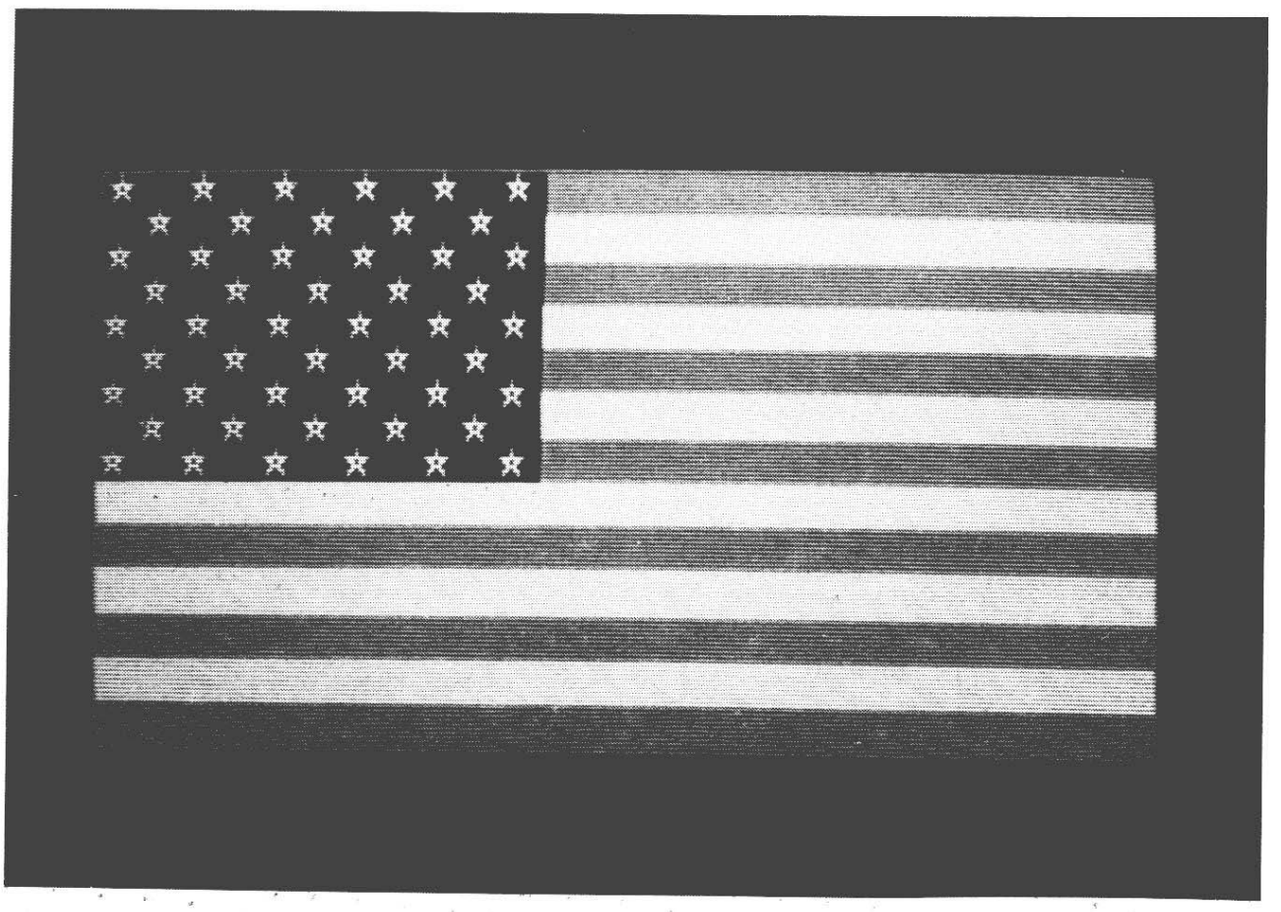
function to differentiate between even- and odd-numbered stripes. And "x0" keeps track of whether we start at position 0 or at position 11.

Next we draw the blue area. And then we draw fifty stars. This is similar to the program STARS. But the stars are now one-quarter size, and we need a more complicated calculation as to where to draw them. We again use mod and a variable x0 to alternate rows of six stars with rows of five. Finally we "frame" the flag, to make it stand out against the background.

Figure 7.6 is the flag as drawn on an IBM PC. Because of the limitation of color choices on the PC, we used a slightly modified version of the program.

```
FLAG

REM Draw U.S. flag

SET WINDOW 0,27,0,13
CLEAR
```

FIGURE 7.6



```
LET x0 = 0
FOR y = 0 to 12                    ! 13 bars
     IF y = 6 then LET x0 = 11
     IF mod(y,2) = 1 then
        SET COLOR "white"
     ELSE
        SET COLOR "red"
     END IF
     BOX area x0,27,y,y+1
NEXT y

SET COLOR "blue"                   ! Blue background for stars
BOX area 0,11,6,13

LET delta = 7/9
SET COLOR "white"
FOR row = 1 to 9                   ! 50 stars
     LET y = (row-.5)*delta + 6
     IF mod(row,2) = 1 then LET x0 = 1 else LET x0 = 2
     FOR col = x0 to 11 step 2
          DRAW star with scale(1/4) * shift(col-.5,y)
     NEXT col
NEXT row

SET COLOR "green"                  ! Frame it
BOX AREA 0,27,0,13

PICTURE star                       ! Picture of a star
     OPTION angle degrees
     LET new = 90
     FOR i = 1 to 5
          LET old = new
          LET new = old + 2/5*360
          PLOT cos(old), sin(old); cos(new), sin(new)
     NEXT i
END PICTURE

END
```

The next program is very short, but it illustrates recursion, a powerful programming technique not available in the more elementary languages.

The greatest common divisor (GCD) of two integers is the largest integer that divides both. Thus, since 100 and 64 are both divisible by 4, but by no larger number, their GCD is 4.

The program finds the GCD of a and b. If b happens to divide into a exactly, then b itself must be the GCD. If not, then we find the remainder r when a is divided by b (using "mod"). The problem is then reduced to finding the GCD of b and r. The defined function "gcd" carries this out. Note that in its definition "gcd(b,r)" occurs. A function or subroutine that calls itself is "recursive." It must be emphasized that this definition

is not circular. It is designed to compute the GCD of larger numbers in terms of the GCD of smaller ones.

```
GCD

REM Find greatest common divisor

DEF gcd(a,b)     ! Greatest common divisor function
     LET r = mod(a,b)          ! Remainder
     IF r = 0 then
        LET gcd = b            ! b divides a
     ELSE
        LET gcd = gcd(b,r)     ! Recursive call
     END IF
END DEF

PRINT "Two numbers";
INPUT a,b                      ! Ask user for numbers
PRINT "The g.c.d. = "; gcd(a,b)

END

run

Two numbers? 1001,66
The g.c.d. = 11
```

For a more powerful use of recursion we return to the problem of where the sine and the common logarithm intersect. This time we will find numerical solutions.

We are trying to solve the equation

```
sin(x) = log10(x)
```

or we can write it as

```
sin(x) - log10(x) = 0
```

Thus we are looking for values of x such that a given function f(x) is equal to 0. Our program solves this problem for any smooth function f(x) expressible in BASIC.

Suppose that the function has different signs at x = a and x = b. Say f(a) is negative while f(b) is positive. Then there must be a zero in the interval [a,b]. The routine "onezero" finds this to six decimal places. It is both simple and very fast. We let x be the midpoint, and check the sign of f(x). If it is negative, then there is a zero between x and b, otherwise between a and x. In either case we have cut the interval in half. It takes very few steps to narrow the interval to the point where we have the answer to six decimal places. For example, if the original interval was of length one, twenty iterations cut it down to one of length one-millionth.

But what if f(a) and f(b) are both positive or both negative? Then we must try to find a point in between that has a different sign. The routine "change" does this, by picking up to fifty points at random in the interval. If it finds one where f(x) has a different sign, it signals by setting ok = 1. If it fails, we can be fairly sure (but not absolutely certain) that there is no zero in this particular interval.

The recursive routine "roots" finds all the roots. It uses "change" to find a change of sign. If one is found, it calls "onezero" and prints the answer. Then it calls itself twice, once to search to the left and once to the right of the previous zero. It stops when none of the remaining intervals has a change of sign.

The function f(x) is the one discussed above. But it can be changed to any function expressible in BASIC. The main program simply calls "roots," indicating the total interval to be searched. In our case, since the logarithm is not defined for x = 0, we start with .5. And we know from Figure 7.3 that all zeros occur before x = 10. You may wish to compare the answers with Figure 7.3. And you can experiment with other functions.

```
ZERO

REM Find all roots of a function

CALL roots(.5,10)

END

DEF f(x) = sin(x) - log10(x)

SUB roots(a,b)                    ! Recursive root finder
    LET olda = a                  ! Remember starting points
    LET oldb = b
    CALL change(a,b,ok)           ! Change of sign?
    IF ok = 1 then                ! Yes
       CALL onezero(a,b,x)        ! Find one root
       PRINT x                    ! Print it
       CALL roots(olda,x-1e-5)    ! Search to the left
       CALL roots(x+1e-5,oldb)    ! Search to the right
    END IF
END SUB
```

```
SUB onezero(a,b,x)              ! Finds one root
    ! Change of sign in [a,b]
    DECLARE FUNCTION f
    LET sign = sgn(f(a))        ! Sign on left
    DO while abs(b-a) > 1e-6    ! To 6 decimals
       LET x = (a+b)/2          ! Midpoint
       IF sgn(f(x)) = sign then
          LET a = x             ! Move left end
       ELSE
          LET b = x             ! Move right end
       END IF
    LOOP
END SUB


SUB change(a,b,ok)              ! Look for change of sign
    DECLARE FUNCTION f
    LET sign = sgn(f(a))        ! Left end
    LET ok = 1
    IF sgn(f(b)) <> sign then EXIT SUB ! Already have
                                                    change
    FOR n = 1 to 50     ! Try up to 100 random points
        LET c = a + (b-a)*rnd
        IF sgn(f(c)) <> sign then EXIT FOR ! Found one
    NEXT n
    IF n>50 then                ! None found
       LET ok = 0
    ELSE                        ! Change in [a,c]
       LET b = c
    END IF
END SUB


run
2.69626
7.32835
8.26383
```

We will build a small library, containing four useful subroutines. They include a routine to open a file with "protection," so that the user can correct errors in supplying the name of the file. Next we have routines to copy the lines of a file to a string list and to copy a string list to a file. Finally, we supply a routine to sort strings.

The user needs only know that the library is in a file called USEFUL, and the name of each SUB and what parameters it takes. We can write a program to alphabetize a file as follows:

```
FILESORT

REM Alphabetize a file

LIBRARY "USEFUL"                ! Utility routines

DIM name$(300)

CALL fileopen(#1)               ! What file?
CALL f_to_l(#1,name$,n)         ! Copy names to list
CALL sort(name$,n)              ! Sort list
ERASE #1                        ! Empty file
CALL l_to_f(name$,#1,n)         ! Copy list to file

END
```

The strategy is very simple. We open a file, we read its lines to the list "name$," sort the list, and replace the contents of the file from the list. And we can do this without knowing how the four routines were coded. The LIBRARY statement makes them all available to us.

We include a listing of the library. There is one new feature in "fileopen." The WHEN structure protects us from a "fatal error." If the file we try to open is not there or not available, control passes to the USE portion. An error message is printed and we loop back for another try. If we do succeed, "ok" is set to 1, and we leave the loop.

The next two subroutines are short and easy to read. And "sort" is one of the standard routines for sorting. It is by no means the fastest one. If one replaced it by a better one in USEFUL, no change would be necessary in FILESORT—it would just run faster.

Libraries are powerful means of building up one's collection of available routines. It is also a means of acquiring routines others have written. True BASIC even allows "compiled" libraries, that is, routines that have already been translated from BASIC. This makes them rapidly available to any program. Such libraries add enormous power to the language.

```
USEFUL

EXTERNAL                              ! Utility routines

SUB fileopen(#1)                      ! Protected open of a file
    LET ok = 0                        ! Not open yet
    DO
       PRINT "File name";             ! Ask user
       INPUT f$
       WHEN error in                  ! Protect
             OPEN #1: name f$
             LET ok = 1               ! Success
       USE
             PRINT "Can't open. Try again."     ! Failure
       END WHEN
    LOOP until ok = 1
END SUB

SUB f_to_l(#1,list$(),n)              ! File to list
    LET n = 0                         ! Count lines
    DO while more #1
       LET n = n + 1
       LINE INPUT #1: list$(n)        ! Input to list
    LOOP
END SUB

SUB l_to_f(list$(),#1,n)              ! n lines from list to file
    FOR i = 1 to n
        PRINT #1: list$(i)
    NEXT i
END SUB

SUB sort(list$(),n)                   ! Alphabetize list
    FOR i = 1 to n-1     ! Smallest remaining to position i
        LET x$ = list$(i)
        LET p = i
        FOR j = i+1 to n              ! Look at rest
            IF list$(j) < x$ then
               LET x$ = list$(j)      ! Smaller
               LET p = j
            END IF
        NEXT j
        IF p <> i then                ! Switch
           LET list$(p) = list$(i)
           LET list$(i) = x$
        END IF
    NEXT i
END SUB
```

There are a number of well-known routines that are used to test the power of a language and how conveniently programs can be structured. Our final example will be one of these routines.

In the previous example "sort" is a simple routine for alphabetizing a list. If we replace "list$" by "list" and "x$" by "x" then the same routine serves to arrange a list of numbers in increasing order. BASIC is one of very few languages in which it is so clear that sorting numbers and alphabetizing are really the same routine.

Let us apply "sort" to long lists of numbers. A list of 100 numbers is sorted in four seconds, which is fine. But a list twice as long will take four times as much time, and a list ten times as long will be one hundred times as time-consuming. Although 400 seconds might be bearable for a list of 1000 numbers, a list with 10,000 numbers requires over eleven hours! (The times quoted are for the IBM PC. While the actual times will vary with the computer, the ratios do not.)

You can speed up sorting 200 numbers by first sorting just 100, then sorting the other 100, and finally merging the results. This is the key idea of "mergesort." If the list is too long, cut it in half. Continue cutting until it is short enough to apply "sort" to each piece. Then merge the pieces.

The program MSORT is a library for carrying this out. It contains "sort," the routine we have seen before, modified for numeric sorting. The key procedure is "mergesort," an elegant recursive routine. If the list contains no more than twenty numbers, then it applies "sort." Otherwise it cuts the list in half and applies itself to each half. Then it merges the two halves.

Let us consider a list of 160 numbers. It is too long, hence is cut into two pieces of 80. Again, it is too long, hence each piece is cut into lists with 40 numbers. Then these are cut into 20-number lists. We end up with 8 pieces of 20 numbers. Each of these is sorted by "sort." Then pairs of pieces are merged into lists of length 40. Pairs of these are merged into lists of 80. And the two such lists are merged to give us a complete sorting of the 160 numbers.

That sounds terribly complicated. The reason is that BASIC is much more efficient at explaining intricate procedures than is the English language! Just look at "mergesort." It is much clearer than the previous paragraph. That is why programs written in BASIC have proved to be powerful teaching tools.

The routine "merge" merges two sorted segments of our list into a single sorted segment. It uses the temporary storage "temp" into which the numbers are moved in the right order. We will not explain the routine; we hope you will have fun figuring it out for yourself.

The following output was produced when "mergesort" was applied

to a list of 200 integers generated at random.

```
run

 0   2   21  27  32  32  48  52  64  69
 72  77  78  81  84  88  90  90  92  100
102 105 106 108 117 121 126 131 142 158
167 170 171 172 174 188 190 196 211 217
217 223 238 244 246 247 252 259 265 275
276 279 288 290 295 297 308 312 315 318
330 335 339 348 355 363 365 381 381 382
389 390 391 394 401 409 413 421 432 441
442 442 455 463 467 468 483 483 484 491
493 494 496 501 505 508 514 522 524 528
529 530 531 537 540 542 544 548 549 550
565 573 575 579 581 582 585 588 589 593
600 602 623 625 632 634 637 642 645 650
651 654 657 669 675 680 683 687 692 695
697 713 717 718 727 736 747 748 750 773
778 778 779 783 790 799 803 806 823 824
828 830 846 848 851 852 858 860 862 867
870 873 879 881 882 882 895 896 896 902
913 923 923 927 934 935 936 937 946 948
951 961 962 964 968 979 983 987 989 990

5 secs.
```

It took only five seconds to sort the list. And a list of 1000 is sorted in thirty-three seconds. Even a list of 10,000 can be sorted in eight minutes, if there is sufficient memory. That is a far cry from the eleven hours required by an ordinary sort.

```
MSORT

REM Library for sorting

EXTERNAL

SUB mergesort(list( ), first, last) ! Recursive mergesort

IF last - first < 20 then ! Use ordinary sort
   CALL sort(list, first, last)
ELSE                              ! Cut in half

   LET mid = int((first + last)/2)
   CALL mergesort(list, first, mid)
   CALL mergesort(list, mid+1, last)
   CALL merge(list, first, mid, last)
END IF


END SUB
```

```
SUB merge (list(), first, mid, last) ! Merge two portions of
                                                        list

    DIM temp(1000)
                                ! Pointer first part
    LET p1 = first
    LET end1 = mid              ! End first part
    LET p2 = mid+1              ! Pointer second part
    LET end2 = last             ! End second part
    LET k = 1                   ! Pointer in temp()

    DO while p1 <= end1 and p2 <= end2 ! Merge two
                                                        parts
       LET x1 = list(p1)
       LET x2 = list(p2)
       IF x1 <= x2 then
          LET temp(k) = x1 ! Move from first
          LET p1 = p1 + 1
       ELSE
          LET temp(k) = x2 ! Move from second
          LET p2 = p2 + 1
       END IF
       LET k = k + 1
    LOOP

    IF p1 <= end1 then
       CALL copy(list,temp,p1,end1,k) ! Copy rest of first
                                                        part
    ELSE
       CALL copy(list,temp,p2,end2,k) ! Copy rest of second
                                                        part
    END IF
    LET where = first
    CALL copy(temp,list,1,k-1,where) ! Copy result to
                                                        list()

END SUB

SUB copy(x(),y(),x1,x2,j) ! Copy list x, x1 to x2, to j at y1

    FOR i = x1 to x2
        LET y(j) = x(i)
        LET j = j+1
    NEXT i

END SUB
```

```
SUB sort(list( ),first,last)      ! Sort list

    FOR i = first to last-1 ! Smallest remaining to
                                                    position i
        LET x = list(i)
        LET p = i
        FOR j = i+1 to last  ! Look at rest
            IF list(j) < x then
               LET x = list(j)     ! Smaller
               LET p = j
            END IF
        NEXT j
        IF p <> i then        ! Switch
           LET list(p) = list(i)
           LET list(i) = x
        END IF
    NEXT i

END SUB
```

NOTES

Introduction

1. T. E. Kurtz, "BASIC Session" in R. L. Wexelblat (ed), *History of Programming Languages* (Academic Press, 1981), 515–549.

Chapter 2

1. ANSI, *American National Standard for the Programming Language Minimal BASIC*, X3.60-1978, New York, 1978.
2. Dartmouth College, *BASIC Instruction Manual*, June 1964.
3. For further details on the design, see J. G. Kemeny and T. E. Kurtz, "Dartmouth Time Sharing," *Science* 162 (1968): 223–228.
4. Dartmouth College, *BASIC Instruction Manual*, June 1964, p. 34.
5. Dartmouth College, *The IMPRESS Manual*, Project IMPRESS, July 1972.
6. R. J. Miara, J. A. Musselman, J. A. Navarro, and Ben Shneiderman, "Program Indentation and Comprehensibility," *Communications of the ACM* 26, no. 11 (November 1983): 861–867.

Chapter 3

1. E. Dijkstra, "Goto Statement Considered Harmful," *Communications of the ACM* 11, no. 3 (March 1968): 147–148.
2. Alan Kay, "A Personal Computer for Children of All Ages," Proceedings of the ACM National Conference, 1972.
3. F. Sykes, R. T. Tillman, and Ben Shneiderman, "The Effect of Scope Delimiters on Program Comprehension," *Software—Practice & Experience* 13 (1983): 817–824.
4. E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," *Communications of the ACM* 26, no. 11 (1983): 853–860.
5. *Draft Preliminary American National Standard for the Programming Language BASIC*, document X3.113, X3 Secretariat, CBEMA, Suite 500, 311 First Street NW, Washington, DC 20001.
6. *The True BASIC Reference Manual* (Addison-Wesley, 1985).

Chapter 6

1. See J. M. Nevison, *The Little Book of BASIC Style* (Addison-Wesley, 1978).
2. S. Papert, *Mindstorms* (Basic Books, 1980).
3. *American National Standard Pascal Computer Programming Language* (IEEE, 1983).
4. ANSI, *Reference Manual for the Ada Programming Language*, (New York, 1983).

INDEX

Ada®, 101–102, 103, 104, 105, 106
ALGOL, 6, 7, 8, 9
American National Standard BASIC, 49–53
American Standard Code for Information Interchange (ASCII), 29, 30
Angles, 121
ANSI Minimal BASIC, 20, 27, 28, 61
Arrays, 20–21, 26, 43, 104, 113
See also MAT statement
ASC function, 30
ASCII (American Standard Code for Information Interchange), 29, 30
Assembly language, 6
Assigned statements, 102–103

BASIC:
Ada® vs., 101–102, 103, 104, 105, 106
COBOL vs., 101, 103, 104, 105, 106
Dartmouth, 13, 14, 19–23, 42, 43, 55
development of, 5–17
FORTRAN vs., 99–101, 103, 104, 105, 106
growth of, 19–37
LOGO vs., 90–93, 103, 104, 105, 106
Minimal, 20, 27, 40, 61, 65
Modern, 40
Pascal vs., 93–99, 103, 104, 105, 106
problems with, 55–66
recent changes in, 39–53
SBASIC, 44–46
Standard, 40, 44, 46, 50–52, 61, 67, 83, 89
standardization of, 65
variations of, 55–56
See also True BASIC; True BASIC programs
BASICA, 63–64

BASIC Standard Committee, 27
Batch processing, 1–3
Beta testing, 36
Branching, 73–77
Busch, Mike, 5

CALL statement, 34–35, 43, 81, 83, 87
Call-360 BASIC, 24
CHANGE statement, 29–30
Character strings, 28–30
CHR$ function, 30
COBOL, 101, 103, 104, 105, 106
Colors, 62, 63, 122–125
COMMON, chaining with, 33
Compilers, 12, 13, 15–16, 17, 52
Computing, interactive, 11–13
CONTINUE, 44, 45, 48
CRTs, 39, 40

Darsimco (Dartmouth Simplified Code), 7
Dartmouth:
  development of BASIC at, 3–17
  General Electric and, 21–23
Dartmouth BASIC, 13, 14, 19–23, 42, 43, 55
Dartmouth Structured BASIC®, 13
Dartmouth Time Sharing System (DTSS), 23
DATA statement, 28, 31, 113, 120
Data structures, 104
Declaration statements, 96, 97, 103–104
DEF, 20, 84–85
Default lower bound, 26–28
Default option, 11
Defined functions, 84–85
Dickey, John Sloan, 4
Dijkstra, Edsgar, 39–40
DIM statement, 26–28
DO loops, 8, 47–48, 61, 70, 77, 79, 82, 114, 115
DOPE (Dartmouth Oversimplified Programming Experiment), 8
DRAW, 120, 122
DTSS (Dartmouth Time Sharing System), 23

ELSE, 75
ELSEIF, 75
Error messages, 12, 13, 52, 61
Error reporting, problems with, 57, 60–61
Exception handling, 53
EXIT DO, 49, 95
EXTERNAL, 87

Federation of Information Processing Societies, 5
Files, 31–32, 111
FILES statement, 32
FINISH, 42, 43
Flag, graphics program for, 123–124
Floating point numbers, 13–14
Formatting, 11
FOR-NEXT loop, 9, 35–36, 49, 61, 68–70, 108–109
FORTRAN, 6–7, 8–9, 11, 13
  True BASIC vs., 99–101, 103, 104, 105, 106
Functions, defined, 84–85

Garland, Steve, 7, 43–44, 48
General Electric, 21–23
General loop, 44

GOSUB, 32–35, 44, 80, 81, 82, 83
GOTO statement, 43, 47, 70, 74, 75, 76, 83
Graphics, 51
  problems with, 62–64
  in various languages, 105
  See also LOGO; Plotting
Graphics programs, 119–124
Greatest common divisor (GCD), program for finding, 125–126
Guessing game program, 114–119

Hargraves, Bob, 7
Hexadecimal number system, 62

IBM BASICA, 63–64
IF statement, 73–77, 111
IF-THEN-ELSE, 44, 45
Indentation, 35–36, 77
INPUT, 12, 25–26, 32, 33, 48
Interactive computing, 11–13
Interpreters, 13, 56–59
ISO Pascal, 98–99

Kay, Alan, 40
Kemeny, John, 2, 3–4, 5–6, 7–8, 14, 15–16
Keywords, 57–58
Knapp, Anthony, 7
Kurtz, Tom, 2–3, 4, 7, 8–9, 14–15, 16–17

Languages:
  Ada®, 101–102, 103, 104, 105, 106
  ALGOL, 6, 7, 8, 9
  assembly, 6
  COBOL, 101, 103, 104, 105, 106
  Darsimco, 7
  DOPE, 8
  FORTRAN, 6–7, 8–9, 11, 13, 99–101, 103, 104, 105, 106
  LOGO, 90–93, 103, 104, 105, 106
  machine, 6
  Pascal, 8, 46, 52, 93–99, 103, 104, 105, 106
  PL/1, 45
  SAP, 7
  Scalp, 7
  See also BASIC; True BASIC
LEFT$, 29
LEN, 29, 30
LET statement, 34, 59–60
Libraries, 86–88, 129–130, 131
LIBRARY statement, 42, 86, 129
LINE INPUT, 26
Line numbers, 11–12, 82–83
LINPUT, 26
Lists, 26
Llacer, Jorge, 7
LOGO, 90–93, 103, 104, 105, 106
Loops, 8, 9, 35–36
  DO, 8, 47–48, 61, 70, 77, 79, 82, 114, 115
  FOR-NEXT, 9, 35–36, 49, 61, 68–70, 108–109
  in FORTRAN, 99–100
  general, 44
  in Pascal, 93–96
  problems with, 61
  in SBASIC, 44–49
Luehrmann, Arthur, 40, 41

McCarthy, John, 3
McGeachie, John, 5

Machine language, 6
Marshall, Sidney, 8
MAT statement, 26–28, 36, 105, 113
Memory, limited, 56, 59–61
Mergesort, 131–134
MID$, 29
Minimal BASIC, 20, 27, 40, 61, 65
Modern BASIC, 40
Modules, 43
MSORT, 131–134

National Science Foundation (NSF), 5
Numbers, 13–14, 62
  floating point, 13–14
  greatest common divisor of, 125–126
  hexadecimal, 62
  in Pascal, 105
  prime, 110–111
  sorting, 131–134

ON statement, 78, 79, 82, 83
OPEN, 111
OPTION, 121
OPTION BASE, 27–28

Parameters, 85–86, 97–98
Pascal, 8, 46, 52
  True BASIC vs., 93–99, 103, 104, 105, 106
Pi, calculating value of, 107–110
PICTURE, 91, 120, 121, 122
PLOT, 42, 43, 119
PLOTTER, 43
Plotting, 39, 40–43
  See also Graphics
POS, 29
Preprocessor, 42–43, 44
Prettyprinting, 25–26
Prime numbers, 110–111
Printing format, 11
PRINT statement, 23–25
PRINT USING, 24
Process-read loop, 48–49
p-system Pascal, 52

Random-access files, 31
Random number generation, 78
Read-process loop, 48–49
READ statement, 113
Recursion, 125–127
RETURN statement, 33–34
Rieser, Leonard M., 4
RIGHT$, 29
Roots, 74, 127
Routines:
  library of, 86–88, 129–130, 131
  string, 86–88
  See also Subroutines

SAP (Symbolic Assembly Program), 7
SBASIC, 44–46
Scalp, 7
SEG$, 29, 30
SELECT, 44, 45, 79, 82, 83
Semicolon, 24
SET COLOR, 62, 119, 124–125
SET WINDOW, 62, 119, 120, 122, 123, 124
Sorting, 131–134
Standard BASIC, 40, 44, 46, 50–52, 61, 67, 83, 89
Standardization, problems with, 65
Strings, 28–30, 86–88
STR$ function, 24

Structured BASIC, 44–46
Structured constructs, 103
Structured programming, 43–49, 67–88
  branching, 73–77
  defined functions, 90–92
  DO loops, 70–77, 79
  FOR-NEXT loops, 68–70
  GOTO, 83
  IF statement, 73–77
  LIBRARY, 86–88
  line numbers, 77, 82–83
  SELECT, 79–83
  subroutines, 83–84
Style, problems with, 64–65
Subroutines:
  early, 20
  growth of, 33–35
  library of, 86–88, 129–130, 131
  named, 45
  in SBASIC, 45
  in structured programming, 83–84
  in True BASIC, 85
  in various languages, 104
Subscript errors, 100–101
Symbolic Assembly Program (SAP), 7

Tables, 26
Terminal-format files, 31
Time-sharing system:
  development of, 3–6
  interactive computing in, 11–13
  separate computers in, 25
Tribus, Myron, 4
True BASIC, 52–53, 89, 105–106
  Ada® vs., 101–102, 103, 104, 105, 106
  advantages of, 67
  branching, 73–77
  COBOL vs., 101, 103, 104, 105, 106
  FORTRAN vs., 99–101, 103, 104, 105, 106
  GOTO, 83
  graphics in, 62–64
  line numbers, 83
  LOGO vs., 90–93, 103, 104, 105, 106
  loops, 72–73
  Pascal vs., 93–99, 103, 104, 105, 106
  subroutines, 85
True BASIC programs:
  counting words in file, 111–113
  graphics, 119–124
  greatest common divisor (GCD), 125–126
  guessing game, 114–119
  prime numbers, 110–111
  roots, 127
  sorting, 131–134
  subroutine library, 129–130, 131
  value of pi, 107–110
Turtle graphics. *See* LOGO

Variable names, 58
Vectors, 26

WHEN structure, 143
WHILE, 60
WINDOW, 43

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "QkFDSyBUTyBCQVNJQ180MDM1ODY2OC56aXA=",
  "filename_decoded": "BACK TO BASIC_40358668.zip",
  "filesize": 51241199,
  "md5": "f05c5ca5130643c03d333905ffbdcf6f",
  "header_md5": "d9a5e56dbe244a1f2f4f479e19450d4c",
  "sha1": "6b9c35bb4570bdbc2db65f03274ce773811da812",
  "sha256": "b20cb167d97a4477c81a7bcc177a77e71784ef2aae131173e17350ca01d1b7af",
  "crc32": 381996131,
  "zip_password": "",
  "uncompressed_size": 58195969,
  "pdg_dir_name": "BACK TO BASIC_40358668",
  "pdg_main_pages_found": 141,
  "pdg_main_pages_max": 141,
  "total_pages": 152,
  "total_pixels": 645929232,
  "pdf_generation_missing_pages": false
}